LES GUIDES DE
GNU LINUX MAGAZINE / FRANCE
HORS-SÉRIE N°87
France MÉTRO. : 12.90 € — CH : 18,00 CHF — BEL/PORT.CONT : 13,90 € — DOM TOM : 13,90 € — CAN : 18,00 $ CAD

KERNEL

LE GUIDE POUR PLONGER AU CŒUR DE VOTRE SYSTÈME GNU/LINUX

DÉBUTEZ
Recompilez et adaptez le noyau à votre PC

EXPÉRIMENTEZ
5 cas concrets de développement noyau

VIRTUALISEZ
Étudiez le fonctionnement de l'hyperviseur KVM

SÉCURISEZ
Utilisez SELinux pour sécuriser l'accès à vos services

Édité par Les Éditions Diamond
L 15066 - 87 H - F: 12,90 € - RD
www.ed-diamond.com

Retrouvez toutes nos publications
LES ÉDITIONS DIAMOND
sur **www.ed-diamond.com**

GNU/Linux Magazine Hors-Série
est édité par **Les Éditions Diamond**

10, Place de la Cathédrale - 68000 Colmar - France

Tél. : 03 67 10 00 20 / **Fax :** 03 67 10 00 21

E-mail : cial@ed-diamond.com
lecteurs@gnulinuxmag.com

Service commercial : abo@gnulinuxmag.com

Sites : http://www.gnulinuxmag.com
http://www.ed-diamond.com

Directeur de publication : Arnaud Metzler

Chef des rédactions : Denis Bodor

Rédacteur en chef : Tristan Colombo

Conception graphique : Kathrin Scali & Thomas Pichon

Responsable publicité : Tél. : 03 67 10 00 27

Service abonnement : Tél. : 03 67 10 00 20

Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne

Distribution France :
(uniquement pour les dépositaires de presse)

MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04

Service des ventes :
Distri-médias : Tél. : 05 34 52 34 01

IMPRIMÉ en Allemagne - PRINTED in Germany

Dépôt légal : A parution

N° ISSN : 0183-0864

Commission Paritaire : K78 976

Périodicité : Bimestrielle

Prix de vente : 12,90 Euros

PRÉFACE

Le noyau est au cœur des systèmes d'exploitation. De tous les systèmes d'exploitation ? Oui, de tous, même de ce système qui résiste encore et toujours aux envahisseurs propriétaires, ce système que nous utilisons tous : GNU/Linux. Bien entendu, nous parlerons dans ce hors-série de ce noyau (ou *kernel* pour ceux qui préfèrent conserver la version non traduite) et délaisserons les noyaux des autres systèmes d'exploitation. Signalons par la même occasion la sortie prochaine du noyau 4.9 annoncée par Linus Torvalds le 15 octobre **[1]**.

Le noyau est un élément du système tout à la fois fascinant et terrifiant. Il nous attire irrésistiblement, car c'est lui qui va gérer les accès au matériel, permettre une communication entre les différents composants et donc au final permettre à l'ordinateur de fonctionner correctement. On voudrait donc instinctivement en savoir plus, comprendre ses mécanismes internes et pouvoir manipuler ce magnifique « jouet ». En même temps, il nous terrifie par la complexité qui en émane et par le fait que commettre une erreur en manipulant le noyau signifie bien souvent la « mort » du système d'exploitation et donc l'impossibilité d'utiliser l'ordinateur...

Le but de ce hors-série est de vous donner les connaissances et les outils nécessaires pour interagir avec le noyau et, en quelque sorte, de faire tomber les barrières qui vous paralysent, qui vous laissent dans la peur. Vous pourrez ainsi :

⇨ compiler le noyau suivant vos propres besoins ;

⇨ écrire du code pour le noyau et le débugger ;

⇨ maîtriser les ressources mutualisées du système d'exploitation et les appels système ;

⇨ utiliser l'hyperviseur KVM ;

⇨ sécuriser l'accès à vos services grâce à SELinux.

Ce que l'on ne connaît pas fait toujours peur. Mais vaincre nos peurs nous libère : après la lecture de ce hors-série vous pourrez manipuler le noyau... à vos risques et périls tout de même ;-)

[1] TORVALDS L., Linux 4.9-rc1 : **http://lkml.iu.edu/hypermail/linux/kernel/1610.1/03881.html**

La rédaction

SOMMAIRE

GNU/Linux Magazine Hors-Série n° 87

1 DÉBUTEZ

Recompilez et adaptez le noyau à votre PC

2 EXPÉRIMENTEZ

5 cas concrets de développement noyau

Kernel

3 VIRTUALISEZ

Étudiez le fonctionnement de l'hypervisseur KVM

4 SÉCURISEZ

Utilisez SELinux pour sécuriser l'accès à vos services

ANNEXE

Retrouvez ici les définitions de certains termes et acronymes présents dans les articles

1

1 DÉBUTEZ

À découvrir dans cette partie...

1.1 Généralités sur le noyau Linux

Le noyau Linux est une des possibilités de fonctionnement d'un système moderne basé sur GNU. Le rôle principal d'un noyau, quel que soit le système d'exploitation, peut être résumé à deux types d'actions : accès au matériel et gestion des priorités. Le noyau Linux est le plus grand projet informatique collaboratif jamais réalisé : plus de 20 millions de lignes de codes écrites par plus de 1500 développeurs. p. 08

1.2 Adaptez le noyau à votre ordinateur

Le noyau est un élément clé dans le fonctionnement d'un système GNU/Linux. C'est un projet extrêmement riche, capable de faire tourner des ordinateurs de bureau ou les plus grands calculateurs possédant des architectures complexes et plusieurs milliers de cœurs. De par sa complexité, peu d'utilisateurs de GNU/Linux osent s'aventure dans la compilation du noyau. Cet article va vous permettre de mieux appréhender les étapes nécessaires à la compilation du noyau. Cela devrait vous fournir les éléments clés pour arriver à obtenir un noyau à façon, adapté à vos besoins et à votre matériel. p. 12

1 DÉBUTEZ

GÉNÉRALITÉS SUR LE NOYAU LINUX

Stéphane TÉLETCHÉA

Bien que les distributions modernes incluent par défaut un noyau fonctionnel pour la plupart des situations, il est toujours intéressant de comprendre son fonctionnement et le rôle qu'il joue dans le système d'exploitation. Cet article va présenter les éléments essentiels d'un noyau GNU/Linux.

Dans sa forme actuelle, le noyau **GNU/Linux** comporte plus de 20 millions de lignes de code, c'est le plus grand projet informatique collaboratif réalisé par l'humanité. Comme tous les noyaux modernes, il doit cependant se plier aux contraintes les plus élémentaires, dictées par le matériel et l'architecture héritée des années 80. Dans cet article, nous allons découvrir les principaux éléments à prendre en compte pour tirer le meilleur de son ordinateur.

1. LA NAISSANCE DE L'ORDINATEUR MODERNE

Dans les années 80, une grande entreprise du nom d'**IBM** a commencé à prendre du retard sur le marché des ordinateurs grands publics. Il existait alors une foison de solutions informatiques qui ont pour la plupart disparu aujourd'hui. Afin de prendre le train en marche, IBM a conçu un ordinateur basé sur quelques principes intangibles : une architecture x86, un système d'initialisation du matériel unique et un seul système d'exploitation **[1]**. Ces contraintes matérielles sont toujours en vigueur à l'heure actuelle, et c'est pour cela qu'il faut toujours un BIOS (un peu plus évolué avec l'apparition de l'**UEFI**) qui aura pour rôle l'initialisation bas niveau du matériel : tension à appliquer, fréquence de fonctionnement, (in)-activation d'un périphérique, priorité des disques durs, etc. **[2]**. Une fois le démarrage bas niveau effectué, le BIOS « passe la main » à un système d'exploitation qui va prendre le relais. C'est là où Linux s'initialise pour fournir à un utilisateur donné l'accès au matériel sans risque **[3]**. Pour permettre un contrôle précis de l'accès au matériel, l'architecture x86 (depuis le 286) dispose d'une organisation en quatre anneaux (voir figure 1).

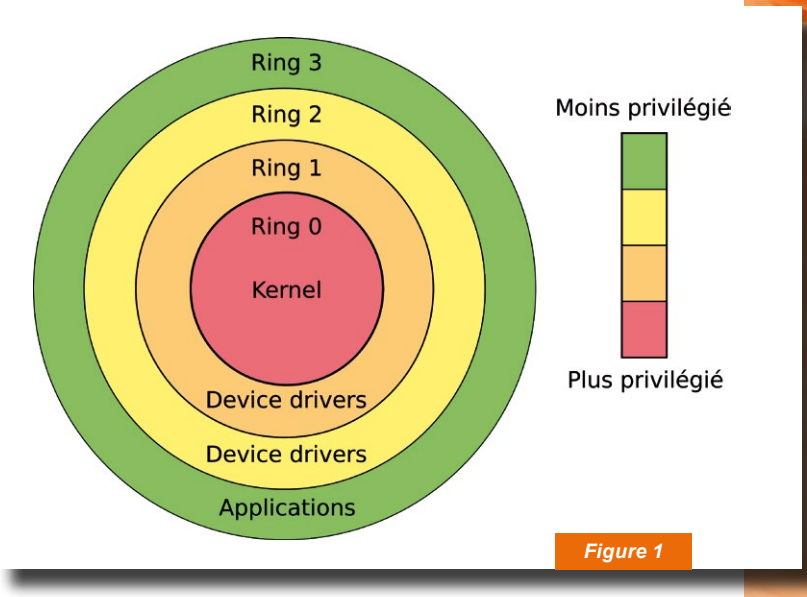


Figure 1

Organisation générale des accès en anneau dans l'architecture x86 (image créée initialement par Hertzsprung sous licence GNU FDL 1.2).

Cette organisation matérielle permet une stricte séparation des privilèges :

- au niveau **0**, seul le noyau peut accéder au matériel pour lire/écrire des données ou modifier son état ;
- les niveaux **1** et **2** sont utilisés pour les périphériques d'entrées/sorties ;
- et seul le niveau de priorité **3** est disponible pour l'espace utilisateur.

Le fonctionnement de l'espace utilisateur est sous contrôle : les ressources mémoires sont bornées et si un dépassement mémoire apparaît, le noyau va prévenir celui-ci, en terminant le processus fautif. En espace privilégié, le noyau a accès à toutes les ressources matérielles, il doit donc présenter un fonctionnement sans faille.

2. ARCHITECTURE GÉNÉRALE ET FONCTIONNEMENT DU NOYAU

Lors de la conception de ce qui allait devenir Linux, **Linus Torvalds** a eu de nombreux échanges à propos de la structure du noyau. Un noyau peut en effet être conçu comme un système minimal qui ne sert qu'à accéder aux périphériques et à gérer les priorités principales (interruptions matérielles, entrées/sorties, etc.). Cette approche dite « micro-noyau » a pour mérite de limiter le code exécuté en espace réservé (*ring 0*), les bugs, et pourrait être idéale dans l'absolu. La taille de ce micro noyau est très réduite, par exemple 5000 lignes seulement de code pour **Minix** ; il est simple et indépendant du matériel, ce qui le rend extrêmement portable. À l'opposé, Linux à ses débuts avait une approche monolithique : tout le code nécessaire était disponible dans le noyau. Cela demande une attention plus importante au code, une plus grande dépendance au matériel. À l'usage cependant, il s'avère que la forme moderne de Linux, monolithique avec architecture modulaire, est plus performante, car son exécution en espace privilégié limite les changements de contexte pour le processeur, et donc permet une meilleure utilisation du matériel *in fine*.

En pratique, la pertinence de l'organisation du noyau dépendra du besoin et de l'architecture qui sera utilisée. Nous verrons dans ce numéro quelles sont les solutions pour faire correspondre au mieux le noyau à son environnement matériel. La plus grosse partie du noyau Linux peut être compilée sous forme monolithique, mais quelques pilotes de périphériques nécessitant des paramètres d'initialisation auront besoin d'être chargés dynamiquement (par exemple certains pilotes WiFI).

Dans tous les cas, le bon fonctionnement du noyau nécessite *a minima* plusieurs fonctionnalités. Les éléments les plus importants sont la gestion de la mémoire, l'accès au système de fichiers et la gestion des processus. La mémoire virtuelle est l'élément fondamental de l'informatique moderne : il s'agit de faire croire à un programme qui le demande, que la mémoire disponible est infinie ; elle est ainsi nommée **mémoire virtuelle**. Si le programme devient trop gourmand en mémoire, en dernier recours, le noyau lui enverra un signal de terminaison (*sigterm*) ou de mort (*sigkill*). De la même manière, dans les systèmes de type **UNIX** sur lesquels Linux a été construit, tout est fichier. Le noyau permet donc d'accéder aux ressources matérielles et logicielles à l'aide d'une couche d'abstraction virtuelle d'un système de fichiers (**VFS**). Les logiciels qui s'exécutent dans un ordinateur sont gérés par le noyau sous forme de processus. Chaque processus possède un code à exécuter, des données à traiter (temporaires), et des ressources en cours d'utilisation ou à allouer. Le noyau doit donc gérer les segments mémoire associés, et attribuer les ressources matérielles nécessaires.

Pour simuler un environnement multi-utilisateur et multi-tâches, il faut que le noyau découpe virtuellement le temps processeur en tranches indivisibles qu'il va répartir de manière régulière aux différentes demandes. Par exemple, si une tâche **A** de compression vidéo s'exécute avec une tâche **B** de recherche sur Internet, le rôle du noyau est d'autoriser **A** à utiliser le processeur et la mémoire pendant *xxx* temps, et d'autoriser **B** pendant *xxx* temps. Nous reviendrons sur les possibilités de découpage plus tard dans le magazine.

Afin de permettre une utilisation complète des capacités matérielles, le noyau va aussi s'appuyer sur les pilotes matériels (*drivers*) via des interfaces génériques dédiées, que ce soit pour l'accès aux périphériques USB ou périphériques réseau. Ces pilotes seront la plupart du temps gérés sous forme de code chargé à la volée (sous forme de module).

3. COMMUNICATION ENTRE L'ESPACE NOYAU ET L'ESPACE UTILISATEUR

Pour permettre une interaction entre le processus qui s'exécute en espace utilisateur avec le plus faible niveau de priorité matérielle et le noyau en espace réservé, Linux offre un ensemble d'appels systèmes regroupés sous le terme de *System Call Interface*. Il existe plus de 300 appels systèmes sous Linux répondant à la norme **POSIX** ou l'étendant **[4]**. Ces appels ont des noms génériques qui permettent de retrouver un comportement identique quel que soit l'objet de l'appel, quelques exemples sont présentés en italique ci-après. Pour la gestion des processus, par exemple, il sera possible de charger du code (*load*), de l'exécuter (*execute*) ou encore de le terminer (*terminate*). La gestion des fichiers (*open*) pourra se faire en lecture (*read*), écriture (*write*) et quand les opérations nécessaires auront été réalisées, le fichier sera fermé (*close*). Pour les périphériques, il faudra dans un premier temps demander l'accès au matériel (*request device*), lire ou écrire des informations (*read*/*write*) puis libérer celui-ci (*release*). Enfin pour les activités réseau, il faudra créer et terminer une connexion (*create* / *delete communication*) et envoyer ou recevoir des données (*send* / *receive*).

Chaque appel normalisé va être reçu par le noyau et en fonction des privilèges attribués au processus demandeur, l'accès à la ressource sera autorisé ou non.

CONCLUSION

Le noyau Linux est très versatile. Après 25 ans de développements, de contributions de bénévoles, de contributions de professionnels et l'arrivée de grands acteurs du marché, il est passé du stade de projet d'un étudiant à celui que nous connaissons aujourd'hui. Nous avons présenté rapidement la genèse du projet et les éléments qui expliquent l'histoire et le développement de Linux. Des contraintes de l'architecture matérielle initiale découlent des impératifs de fonctionnement spécifiques. Grâce à l'expertise acquise dans le monde informatique avant la création de Linux, le fonctionnement du noyau a pu s'appuyer sur des éléments établis pour la gestion du matériel et pour assurer les modalités d'accès aux ressources. D'un système limité à l'ordinateur d'un jeune étudiant isolé, le noyau Linux est maintenant capable de gérer tout type d'architecture matérielle, en offrant à l'utilisateur une interface unique d'accès. ■

RÉFÉRENCES

[1] Une brève histoire du PC : **https://fr.wikipedia.org/wiki/IBM_PC**

[2] L'utilisation du BIOS et ses subtilités sous Ubuntu : **https://doc.ubuntu-fr.org/uefi**

[3] L'architecture x86 en détail : **http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf**

[4] La norme POSIX : **https://fr.wikipedia.org/wiki/POSIX**

POUR ALLER PLUS LOIN

La biographie officielle de Linus Torvalds par David Diamond, épuisée en français, mais toujours disponible sur le site de Framasoft : **https://framasoft.org/article2966.html** ou en anglais « Just for Fun » aux Éditions Eyrolles, 2001, 262p.

ADAPTEZ LE NOYAU À VOTRE ORDINATEUR

Stéphane TÉLETCHÉA

Le noyau fourni pour les distributions GNU/Linux est prévu pour répondre à de multiples usages, il est généraliste. Cet article va vous permettre de compiler un noyau à façon, plus compact et adapté au matériel présent.

Chaque fois que l'on me demande pourquoi compiler son noyau, ma réponse est toujours la même : c'est amusant ! C'est aussi probablement la seule fois où vous aurez affaire à un code aussi versatile et complexe qui présente si peu d'erreurs et de messages d'avertissements. Nous allons voir dans cet article les grandes étapes à suivre pour obtenir son noyau.

1. LES BONNES RAISONS DE COMPILER SON NOYAU

Mis à part le côté ludique, l'autre bonne raison de compiler son noyau c'est bien entendu de maîtriser son environnement. Les noyaux à tout faire des distributions ont ainsi tendance à avoir de nombreuses options activées, mais inutiles pour votre situation, à prendre les paramètres de compilation par défaut au lieu des paramètres optimisés pour votre architecture, ou encore à effectuer des réglages qui ne correspondent pas à votre besoin. Tous ces éléments mis bout à bout font qu'un espace non négligeable est occupé sur le disque pour une utilité discutable.

Il peut être impératif de recompiler votre noyau si une faille de sécurité est identifiée, mais il est difficile de mettre en place un correctif approprié. C'est par exemple ce qui est arrivé il y a quelques années avec une faille liée au chargement de modules dans le noyau GNU/Linux. Une solution temporaire était de désactiver ce fonctionnement à la volée, mais ce réglage n'était pas permanent et devait être à nouveau réalisé après un redémarrage (voir à ce sujet la note d'information de l'agence nationale de la sécurité des systèmes d'information [1]).

Comme les distributions ont pris l'habitude de baser chaque édition sur une version figée du noyau, vous pouvez être limité pour l'utilisation de votre matériel. Dans Ubuntu 16.04 LTS par exemple, il n'existe plus de version propriétaire pour le pilote AMD. Seul le pilote libre peut être utilisé pour l'affichage graphique, or depuis la version 4.4 de nombreuses améliorations en termes de performances et de stabilité ont été apportées [2]. Dans ces conditions, il peut être utile de comparer son noyau actuel avec un noyau optimisé.

Si votre environnement matériel est contraint ou que vous cherchez tout simplement à récupérer de la place disque, la réduction du nombre de modules utilisés et la réduction de la taille du noyau apporteront un gain non négligeable que nous allons illustrer ci-après.

Maintenant que vous êtes convaincus de l'intérêt de la recompilation de son noyau, nous allons pouvoir passer en revue les étapes vous permettant d'y arriver sans encombre.

2. FAIRE CONNAISSANCE AVEC SON MATÉRIEL

Afin d'obtenir un résultat optimal, il faut déterminer les composants nécessaires au bon fonctionnement de son ordinateur. Avant de lancer la configuration des modules et options du noyau, il faut tout d'abord lister ceux déjà présents dans le noyau de la distribution, ce sera un bon point de départ pour connaître le nom des modules associés au matériel. Il faut pour cela utiliser la commande `lsmod`. Pour associer l'ensemble des périphériques, il est possible d'avoir une vue graphique de l'organisation de l'ordinateur avec la commande `lstopo` (paquet `hwloc` sur Debian). Pour lister les périphériques de bus ou les périphériques virtuels, il suffira de les lister avec les commandes commençant par « ls » : `lspci`, `lsusb`, `lshal`, etc. L'ensemble de ces commandes n'est pas indispensable pour faire l'inventaire des noms de modules avec les correspondances

matérielles, mais cela vous permettra de mieux vous familiariser avec les outils disponibles. Vous pourrez compléter les explications présentées ici en vous rendant sur le wiki de Debian [3] .

Sur l'ordinateur utilisé pour la rédaction de l'article, la commande `lstopo` permet d'obtenir le résultat présenté en figure 1.

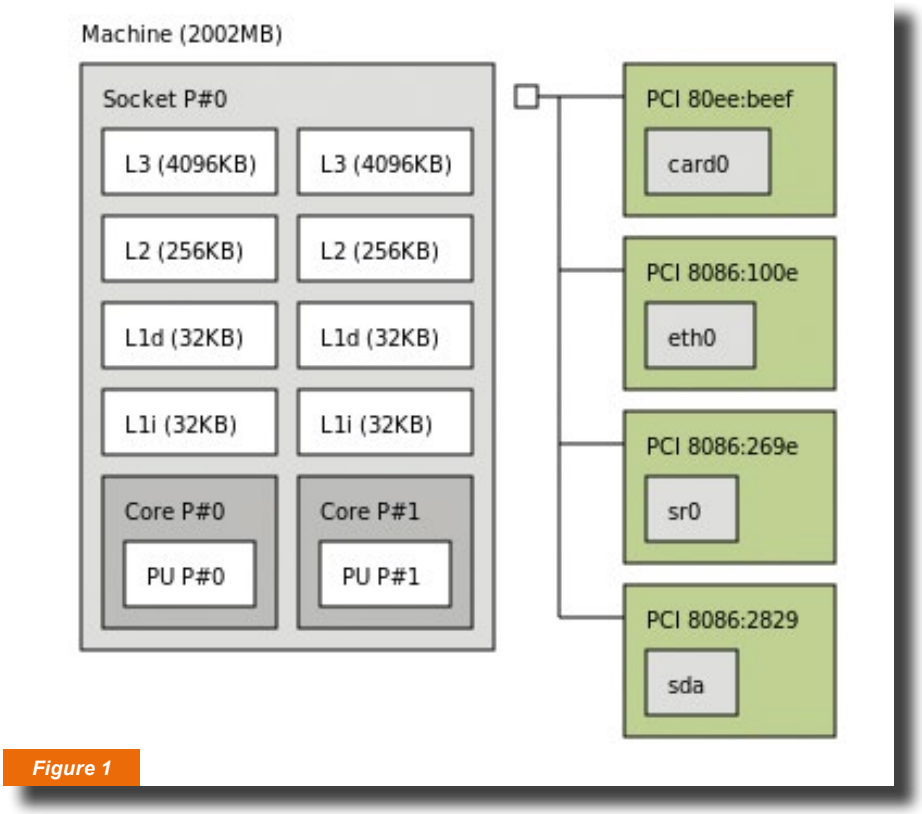


Figure 1

Représentation de l'organisation physique des périphériques en fonction de leur position sur la carte mère.

Un extrait du résultat de `lspci` est disponible dans les lignes suivantes :

Terminal

```
00:00.0 Host bridge: Intel Corporation Core Processor DRAM Controller (rev 02)
00:01.0 PCI bridge: Intel Corporation Core Processor PCI Express x16 Root Port
(rev 02)
00:16.0 Communication controller: Intel Corporation 5 Series/3400 Series
Chipset HECI Controller (rev 06)
00:16.3 Serial controller: Intel Corporation 5 Series/3400 Series Chipset KT
Controller (rev 06)
00:19.0 Ethernet controller: Intel Corporation 82577LM Gigabit Network
Connection (rev 05)
00:1a.0 USB controller: Intel Corporation 5 Series/3400 Series Chipset USB2
Enhanced Host Controller (rev 05)
00:1b.0 Audio device: Intel Corporation 5 Series/3400 Series Chipset High
Definition Audio (rev 05)

00:1d.0 USB controller: Intel Corporation 5 Series/3400 Series Chipset USB2
Enhanced Host Controller (rev 05)
00:1e.0 PCI bridge: Intel Corporation 82801 Mobile PCI Bridge (rev a5)
```

Les éléments présentés concernent les ports hôtes (*bridge*) pour la RAM ou le PCI express, les ports série ou USB ou encore le contrôleur réseau. La révision indiquée entre parenthèses pourra être utile dans certaines options de compilation, en particulier pour les contrôleurs audio s'ils ne sont pas automatiquement détectés. Dans ce cas il faudra mettre le code à charger sous forme de module pour forcer la prise en compte du bon *chipset* lors du chargement pilote.

Prenez le temps de faire l'inventaire de votre matériel, cela vous permettra d'aller plus vite dans les étapes de compilation qui vont suivre.

3. LA COMPILATION À PROPREMENT PARLER

À titre d'illustration, nous allons utiliser le noyau 4.8 comme référence, mais tout autre noyau pourra être compilé de manière identique. Il faut tout d'abord télécharger l'archive du noyau sur le site de **kernel.org** puis la décompresser, et vérifier la provenance du code téléchargé (à quoi servirait de compiler un noyau à façon pour améliorer la sécurité du système sans vérification de l'origine des données...) :

Terminal

```
$ wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.8.tar.xz
$ wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.8.tar.xz.sign
$ gpg --verifify https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-
4.8.tar.xz.sign
$ tar -Jxf linux-4.8.tar.xz
$ cd linux
```

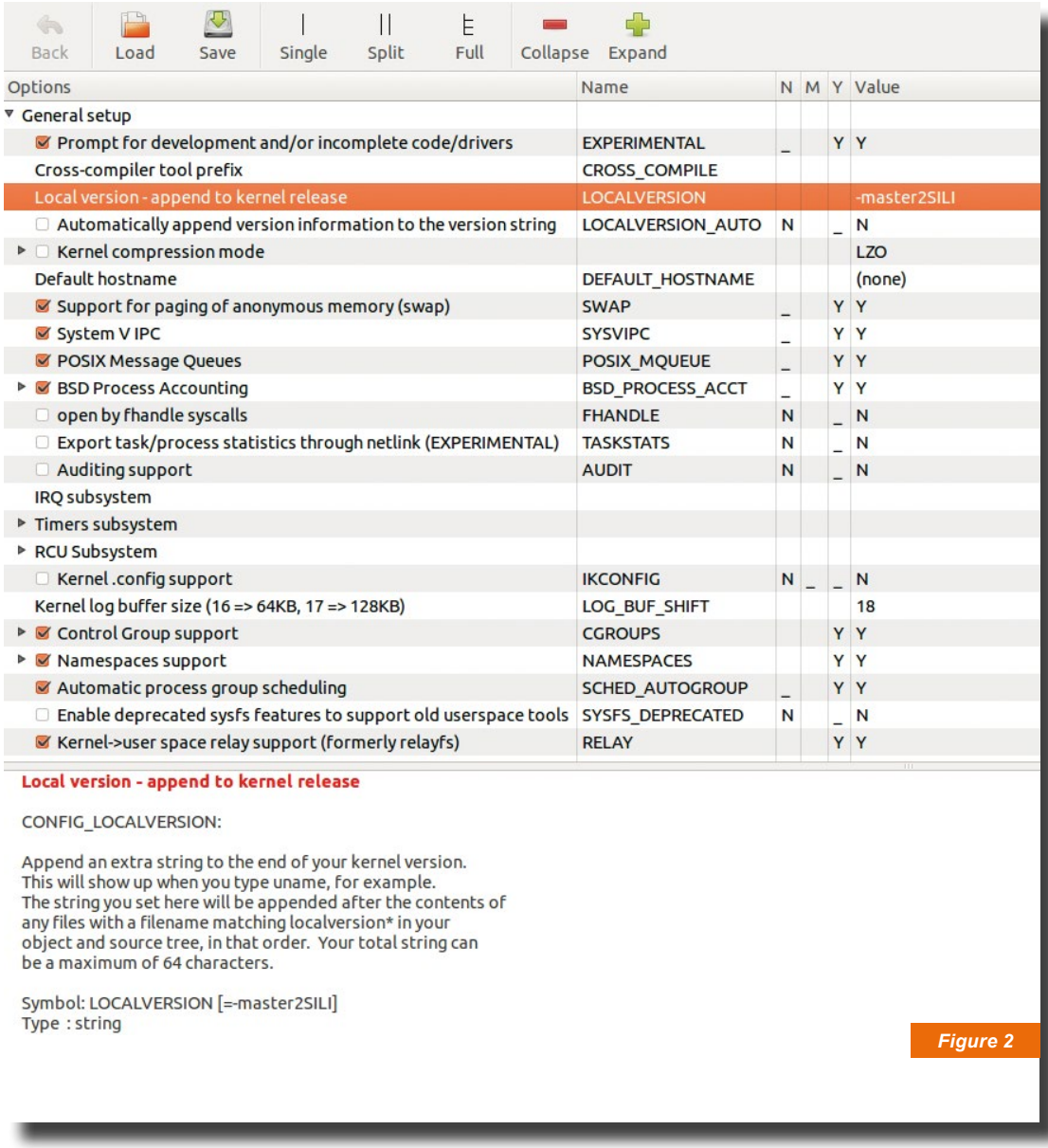
Parmi les fichiers présents dans l'archive, il faut noter que le noyau est en GPLv2 (et le restera) et qu'il y a un fichier **README**. Sa lecture n'est pas inintéressante pour savoir quoi faire pour la compilation du noyau... Mais attention, il faut bien lire l'ensemble du fichier, et non pas les parties qui vous intéressent, en effet sans cette lecture attentive, vous allez faire un **make mrproper** avant de faire la sélection des options. Vous vous retrouverez alors avec une quantité phénoménale d'options à passer en revue.

À la racine de l'archive décompressée, effectuez plutôt un **make defconfig** puis lancez l'interface graphique de sélection des options avec **make gconfig**.
Au besoin, installez au préalable les paquets **libgtk2.0-dev**, **libglade2-dev**, **libglib2.0-dev** ou leurs équivalents dans d'autres distributions que celles basées sur Debian.

L'interface graphique qui est présentée en figure 2 (page suivante) permet de dérouler par simple clic l'arborescence des dépendances entre les modules de compilation. Le noyau Linux est un code complexe, interdépendant, il est donc recommandé de prendre l'interface graphique qui va gérer ces dépendances à votre place. Dans un premier temps, nous pouvons compléter le nom du noyau pour qu'il soit spécifique à votre compilation. Pour accéder à la zone de dialogue, il faut double-cliquer sur celle-ci puis entrer le nom qui vous intéresse, par exemple **-gnulinuxmag**.

Pour bénéficier d'une meilleure gestion du processeur, vous pouvez par exemple sélectionner l'option ***Core 2 Duo*** si votre processeur est de ce type. Au lieu de bénéficier uniquement des instructions x86-64 génériques, vous pourrez ainsi tirer profit

des performances de votre dernier processeur (voir figure 3). En fonction des situations, les performances seront identiques avec les options génériques de compilation, vous verrez à l'usage le gain apporté.
En général, un « ressenti » de meilleure réponse du système est observé, mais il est difficile à quantifier.



Une des premières personnalisations qui peut être réalisée est de modifier le nom de la version locale du noyau.

Nous avons vu précédemment comment faire la liste du matériel de l'ordinateur pour baser la sélection des modules sur le noyau courant. Il faut bien entendu continuer à sélectionner tous les modules nécessaires pour bien initialiser le matériel. En fonction de la situation, vous avez intérêt à mettre le code en l'incorporant dans le noyau pour gagner en rapidité d'exécution, ou en module quand celui-ci va avoir besoin d'une couche logicielle additionnelle. L'exemple caractéristique du besoin d'utilisation en module est pour la gestion du WiFi : comme la configuration sous GNU/Linux peut faire appel à plusieurs couches logicielles, par exemple **`WPA_supplicant`** pour la gestion des clés de connexion, il ne faut pas mettre ce code directement dans le noyau sous peine de ne pas le voir fonctionner.

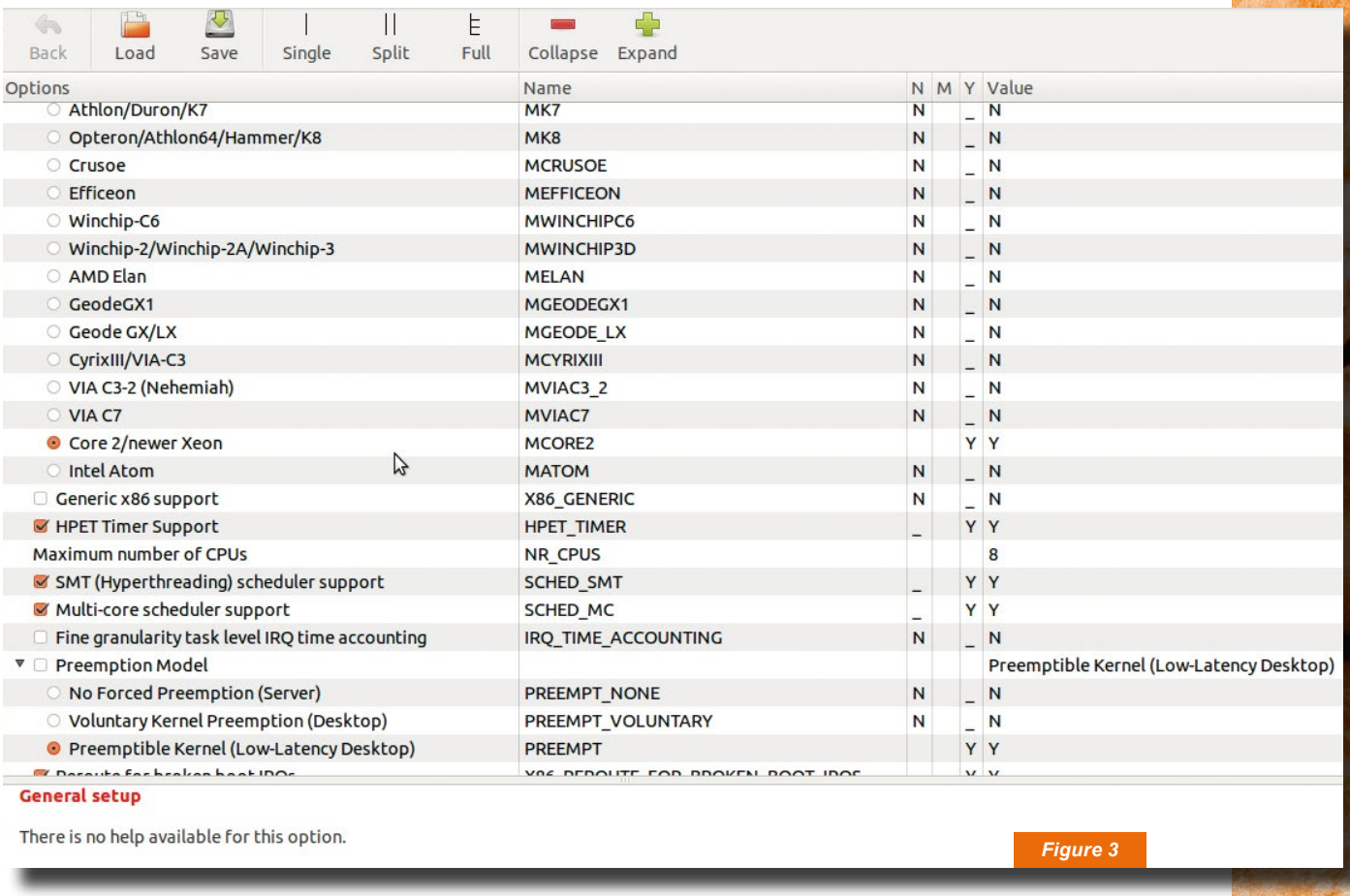


Figure 3

L'option core2Duo peut apporter des gains de performances dans certaines situations.

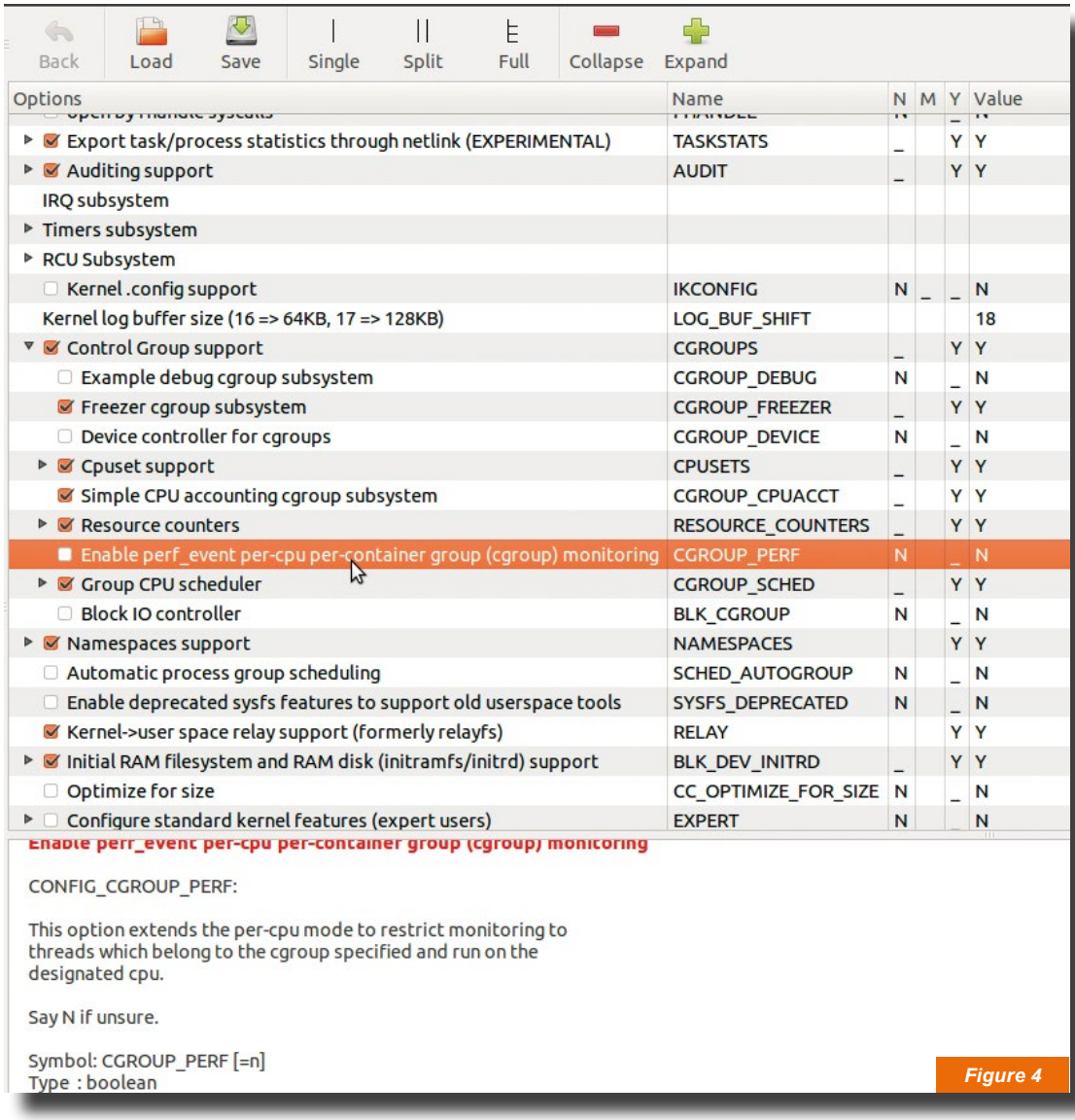
À l'inverse, certaines briques logicielles ont besoin de la présence de fonctionnalités dans le noyau. C'est par exemple le cas pour les **cgroups** (voir article dans le présent hors-série) indispensables pour **systemd** et pour la gestion fine de l'utilisation des ressources disponibles. N'oubliez pas de les activer (voir figure 4 page suivante).

Pour éviter les mauvaises surprises, n'ayez pas la main trop lourde sur la suppression d'options ou sur la transformation de code disponible en module pour un code intégré au noyau. Avant de réussir à avoir un noyau parfaitement fonctionnel pour toutes les situations, il faudra probablement y aller par essais et erreurs. En sus des éléments présentés, il faudra en effet penser aux périphériques connectés temporairement comme une clé USB, une imprimante, une webcam, etc. Avec la diminution du code à compiler, vous pourrez rapidement corriger les éléments manquants sans reprendre à zéro tout le travail réalisé.

Sauvegardez le fichier de configuration, il est temps de passer à la compilation.

4. L'HEURE DU BILAN

Quand la configuration idéale est atteinte, il faut tester si les options choisies sont suffisantes pour initialiser le matériel et apporter tous les éléments logiciels nécessaires au bon fonctionnement de la distribution. Pour installer le noyau et le tester, il ne reste plus que quelques étapes : copier le noyau et les fichiers de configuration dans le répertoire `/boot` et le référencer pour le grub. La plupart de ces options seront effectuées automatiquement dans les distributions basées sur Debian à l'aide de la commande `installkernel`.



Dans une distribution moderne, il est indispensable d'activer certains composants, comme ici les cgroups.

Pour être le plus générique possible, le script suivant présente les commandes qui pourront être utilisées avec toutes les distributions :

Terminal

```
# Tout d'abord il faut compiler le noyau
$ make -j 5 # à adapter en fonction du nombre de processeurs disponibles, en règle générale n processeurs +1
$ sudo make modules_install
$ sudo cp arch/x86/boot/bzImage /boot/vmlinuz-4.8-gnulinuxmag
$ sudo cp System.map /boot/System.map-4.8-gnulinuxmag
$ sudo cp .config /boot/config-4.8-gnulinuxmag
# Il faut aussi créer l'image de démarrage 'initramfs' qui va servir à charger le SE après la décompression du noyau :
$ sudo mkinitramfs -k -o initrd.img-4.8-gnulinuxmag /boot/initrd.img-4.8-gnulinuxmag
# Enfin il faut ajouter le nouveau noyau aux entrées de menu de GRUB
$ sudo update-grub2
```

Au redémarrage, le noyau vous sera présenté parmi les choix possibles, le noyau de la distribution ne sera pas effacé, en cas de pépin vous pourrez toujours revenir à un noyau moins performant, mais fonctionnel...

Au départ, il vous faudra probablement recompiler plusieurs fois le noyau et réaliser les étapes décrites dans les différents scripts plusieurs fois. La procédure décrite est cependant relativement robuste, et avec un peu de patience, vous devriez avoir un noyau fonctionnel, configuré très précisément et très à jour. Au-delà de l'exercice, vous allez commencer à maîtriser un des éléments les plus fondamentaux d'un système GNU/Linux.

CONCLUSION

Nous avons présenté dans cet article les grandes étapes de compilation d'un noyau. En suivant les explications et conseils développés, vous serez heureux d'avoir un noyau fonctionnel au premier redémarrage... Pour trouver des explications sur les erreurs rencontrées ou sur des améliorations que vous pourrez apporter à votre premier tour de compilation, vous pouvez consulter le site d'Ubuntu concernant les étapes à réaliser **[4]** ou encore celui d'ArchLinux, **[5]** pour n'en citer que deux.

Avec un noyau plus compact, le temps de compilation sera aussi réduit, mais ce n'est pas le seul intérêt, car le noyau installé dans `/boot` est compressé, vous aurez ainsi un gain de 1 Mio au maximum sur le noyau seul. Par contre, en ce qui concerne les modules, le répertoire qui permet de les stocker va passer d'une occupation disque de plus de 150 Mio à 15 Mio ou moins. Comparez la taille des répertoires présents dans `/lib/modules` pour savoir le gain réel pour votre situation particulière. Au final le démarrage sera plus rapide, votre ordinateur plus réactif et l'espace disque occupé sera réduit.

La compilation du noyau vous demandera du temps et de la réflexion, mais c'est un exercice très instructif qui vous formera pour la vie. Prévoyez du temps devant vous : la satisfaction que vous allez en tirer sera proportionnelle à l'investissement que vous ferez. ▉

RÉFÉRENCES

[1] Le site national de la sécurité des systèmes d'information :
http://www.ssi.gouv.fr/uploads/2015/10/NP_Linux_Configuration.pdf

[2] Évolution des performances du pilote AMDGPU en fonction des versions du noyau :
http://www.phoronix.com/scan.php?page=search&q=AMDGPU

[3] Une page très complète pour explorer tout le matériel :
https://wiki.debian.org/fr/HowToIdentifyADevice/

[4] Pour aller plus loin avec Ubuntu dans les subtilités des options du noyau :
https://help.ubuntu.com/community/Kernel/Compile#Install_the_new_kernel

[5] Des explications très claires sont aussi fournies sur le site d'ArchLinux :
https://wiki.archlinux.org/index.php/Kernels/Traditional_compilation

POUR ALLER PLUS LOIN

Si après cette découverte du noyau personnalisé vous souhaitez en savoir plus, il est toujours possible de consulter le site des débutants pour le noyau **https://kernelnewbies.org/**. Pour améliorer la réactivité du système, vous pouvez aussi tester l'ordonnanceur de Con Kolivas (**http://ck.kolivas.org/patches/bfs/bfs-faq.txt**).

2

2 EXPÉRIMENTEZ

À découvrir dans cette partie...

CAS N°1

INTERACTIONS ENTRE ESPACE UTILISATEUR, NOYAU ET MATÉRIEL

Christophe BLAESS

Pour pouvoir comprendre ou écrire du code kernel, il est important de bien assimiler les échanges d'informations entre les différents composants du système. Nous examinerons dans cet article les communications et notifications entre l'espace utilisateur (les applications), le noyau Linux et le matériel sous-jacent.

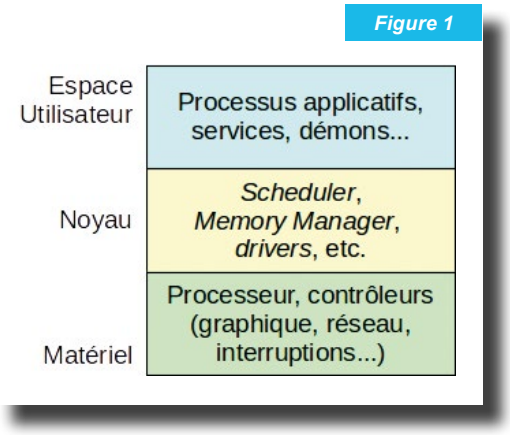
Lorsqu'on aborde la programmation dans le noyau Linux, il n'est pas rare de se sentir un peu submergé par l'ensemble des concepts présents dans ce nouvel environnement. Si l'adaptation d'un driver existant pour supporter un nouveau matériel est souvent assez facile, il n'en reste pas moins que pour écrire du code portable et robuste, certains mécanismes doivent être bien compris. Nous allons en voir quelques-uns dans cet article.

1. STRUCTURE DU SYSTÈME LINUX

Le modèle des systèmes GNU/Linux repose sur trois éléments distincts comme on le voit sur la figure 1.

Le matériel comprend le processeur (composé de diverses unités : CPU, ALU, FPU, MMU, etc.), la mémoire et les périphériques externes au processeur (contrôleur graphique, réseau, son, disque, etc.).

Figure 1



Modèle de système GNU/Linux.

La partie logicielle est divisée en deux niveaux : **l'espace utilisateur** et l'**espace noyau**. Il y a une véritable frontière entre ces deux espaces. En effet, les processeurs actuels possèdent au moins deux modes de fonctionnement : le mode **utilisateur** et le mode **superviseur**. Le code qui s'exécute alors que le processeur est en mode superviseur dispose de tous les droits sur le système. Il peut accéder directement au matériel, activer ou désactiver des interruptions, reconfigurer l'adressage mémoire via la MMU, etc. À l'inverse, un code qui s'exécute en mode utilisateur n'a aucune de ces possibilités.

Le noyau Linux s'exécute en mode superviseur et tout le reste du système en mode utilisateur (y compris les commandes lancées avec `su` ou `sudo` qui n'ont rien à voir avec le mode superviseur du processeur). Nous avons donc deux mondes bien distincts, l'un totalement privilégié (celui du kernel) et l'autre complètement protégé, où aucune erreur de programmation n'aura de conséquence désastreuse sur le matériel ni sur le reste des applications.

Il y a trois manières de passer du mode utilisateur au mode noyau :

- Il existe une liste bien définie de points d'entrée que les programmes peuvent invoquer. Ce sont les **appels système** (par exemple : `open()`, `read()`, `write()`, `ioctl()`, `mmap()`, `fork()`, etc.) dont nous allons étudier le principe ci-après.
- Lorsqu'un périphérique doit notifier le système de l'occurrence d'un événement (par exemple, l'arrivée d'un caractère sur un port série, la fin d'une lecture d'un bloc depuis le disque, la disponibilité d'une trame reçue sur un adaptateur Ethernet, etc.) il fait une demande d'**interruption**. Ce signal électronique indique au processeur d'arrêter son travail en cours, de sauvegarder son contexte d'exécution, et de se brancher à une adresse bien définie où se trouve un code capable de gérer l'événement survenu. Par la même occasion, le processeur bascule en mode superviseur, aussi toutes les fonctions de gestion – les *handlers* – d'interruptions se trouvent dans le noyau Linux.
- Quand un programme de l'espace utilisateur commet une erreur très grave (tentative de division par zéro, déréférencement d'un pointeur invalide, exécution d'un code assembleur illégal, etc.), le processeur détecte directement le problème et notifie le système par un mécanisme proche de celui des interruptions que l'on nomme des **exceptions**. Les *handlers* d'exceptions sont automatiquement exécutés en mode superviseur, et se trouvent donc également dans le kernel. Leur réaction est d'envoyer au processus un signal (`SIGFPE`, `SIGSEGV`, `SIGILL`...) qui va généralement le tuer.

2. LES APPELS SYSTÈME

2.1 Entrée dans le kernel

Comment un appel système – une fonction implémentée dans la bibliothèque C, donc dans l'espace utilisateur – peut-il faire basculer le processeur en mode superviseur et exécuter du code dans le noyau ?

Examinons le fonctionnement de ce petit programme minimal qui invoque l'appel système `write()`.

Fichier

hello.c :

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
        write(STDOUT_FILENO, "Hello\n", 6);
        return 0;
}
```

Nous compilons le fichier source et obtenons un exécutable :

Terminal

```
$ gcc  hello.c  -o hello  -static
```

L'option `-static` est très importante. Elle demande à l'éditeur de liens d'inclure directement dans le fichier exécutable tout le code nécessaire à son fonctionnement, le rendant ainsi indépendant des bibliothèques dynamiques. Nous savons ainsi que tout le contenu de la fonction `write()` dans l'espace utilisateur est maintenant présent dans le fichier. Nous pouvons la désassembler et rechercher comment elle déclenche l'exécution de code dans le noyau. L'exemple a été tout d'abord compilé sur un processeur x86 64 bits.

Terminal

```
$ objdump  -d  hello
[...]
000000000040105e <main>:
  40105e:       55                      push   %rbp
  40105f:       48 89 e5                mov    %rsp,%rbp
  401062:       ba 06 00 00 00          mov    $0x6,%edx
  401067:       be 04 35 49 00          mov    $0x493504,%esi
  40106c:       bf 01 00 00 00          mov    $0x1,%edi
  401071:       e8 1a 2e 03 00          callq  433e90 <__libc_write>
```

Inutile de connaître réellement l'assembleur x86 pour comprendre en substance ce morceau de code. Nous voyons que la fonction `main()` place les trois paramètres pour `write()` dans trois registres avant d'appeler la fonction `__libc_write()`. Comme nous avons compilé notre code avec l'option `-static`, cette dernière est également présente dans le fichier exécutable :

Terminal

```
0000000000433e90 <__libc_write>:

  [...]
  433e99:       b8 01 00 00 00          mov    $0x1,%eax
```

```
433e9e:       0f 05                   syscall
433ea0:       48 3d 01 f0 ff ff       cmp    $0xfffffffffffff001,%rax
433ea6:       0f 83 34 46 00 00       jae    4384e0 <__syscall_error>
433eac:       c3                      retq
```

La fonction `__libc_write()` place la valeur `1` dans le registre `eax` et exécute l'instruction `syscall`. Cette instruction du processeur x86-64 sert à implémenter les points d'entrée du système d'exploitation. Elle va basculer le processeur en mode superviseur et sauter à une adresse donnée où l'on utilisera le contenu du registre `eax` pour sélectionner l'appel système à exécuter. En retour, le registre `rax` contient une valeur négative en cas d'erreur.

En réitérant l'expérience sur un processeur ARM, on observe :

Terminal

```
0001f0f0 <__libc_write>:
   [...]
   1f104:       e3a07004        mov     r7, #4
   1f108:       ef000000        svc     0x00000000
```

Ici c'est le registre `r7` qui reçoit le numéro de l'appel-système – variable selon les architectures, il s'agit maintenant de `4` alors que sur x86-64 nous voyions `1`. L'instruction de déclenchement est `svc` (*service call*).

Sur un processeur x86 32 bits, l'opération est un peu plus compliquée, car il y a une indirection supplémentaire. La fonction `__libc_write` contient :

Terminal

```
806ce4b:       8b 54 24 10             mov    0x10(%esp),%edx
806ce4f:       8b 4c 24 0c             mov    0xc(%esp),%ecx
806ce53:       8b 5c 24 08             mov    0x8(%esp),%ebx
806ce57:       b8 04 00 00 00          mov    $0x4,%eax
806ce5c:       ff 15 f0 a9 0e 08       call   *0x80ea9f0
```

La dernière instruction signifie que l'on invoque une sous-routine dont l'adresse est elle-même stockée en mémoire en `0x080ea9f0`. Si on cherche le contenu de cet emplacement mémoire :

Terminal

```
$ objdump  -s  --start-address=0x080ea9f0   hello
 80ea9f0   f0ef0608 90ad0908 07000000 7f030000  ................
 80eaa00   03000000 02000000 00100000 f0790908  .............y..
 [...]
```

Les quatre premiers octets contiennent l'adresse mémoire de la commande de passage en mode superviseur. Comme l'architecture est *little endian*, il faut inverser les octets et lire : `0x0806eff0`. Voyons ce qui se cache à cette adresse :

Terminal

```
$ objdump  -D  --start-address=0x0806eff0  hello
0806eff0 <_dl_sysinfo_int80>:
 806eff0: cd 80                 int    $0x80
 806eff2: c3                    ret
```

Sur un PC 32 bits, c'est donc une interruption logicielle déclenchée par le CPU lui-même (instruction assembleur `int`), qui permet de passer en mode superviseur et d'exécuter l'appel-système dont le numéro est stocké dans le registre `eax`. Le même numéro d'interruption – `0x80` – est employé pour tous les appels système.

Les lecteurs ayant utilisé MS-DOS dans les années 90 se souviendront peut-être de l'interruption logicielle `0x21` qui permettait en suivant le même principe d'accéder depuis un programme en assembleur aux services du système d'exploitation. Ce fonctionnement est principalement employé sur les anciens processeurs, car les plus récents utilisent de préférence un mécanisme proche de celui des 64 bits.

`syscall` sur processeur x86-64, `svc` sur processeur Arm ou `int 0x80` sur x86-32 ont le même objectif : passer en mode superviseur et se brancher sur une routine dans le noyau. Cette routine est codée en assembleur. Sur une architecture x86-64, il s'agit de `entry_SYSCALL_64` dans le fichier `linux-4.7/arch/x86/entry/entry_64.S`, de `entry_SYSENTER_32` dans `linux-4.7/arch/x86/entry/entry_32.S` pour les x86-32 bits et de `vector_swi` dans `linux-4.7/arch/arm/kernel/entry-common.S` pour les processeurs ARM. Le principe consiste alors à chercher l'adresse de la routine implémentant l'appel système demandé dans une table (la `sys_call_table` définie dans `linux-4.7/arch/x86/entry/syscalls/syscall_32.tbl` ou `syscall_64.tbl` ou encore `linux-4.7/arch/arm/kernel/calls.S`), puis à invoquer cette fonction, par exemple :

Fichier

```
call    *sys_call_table(, %rax, 8)
```

La valeur `8` ci-dessus correspond à la taille des éléments de la table sur cette architecture, 8 octets soit 64 bits. Voici un aperçu du fichier `linux-4.7/arch/x86/entry/syscalls/syscall_64.tbl` qui sert à construire cette table :

Fichier

```
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    sys_read
1       common  write                   sys_write
2       common  open                    sys_open
3       common  close                   sys_close
4       common  stat                    sys_newstat
5       common  fstat                   sys_newfstat
```

Nous voyons que le numéro de l'appel `write()` est bien `1` sur cette architecture comme nous l'avions observé en désassemblant l'appel système. La fonction exécutée sera `sys_write()` qui se trouve dans `linux-4.7/fs/read_write.c` et est commune à toutes les architectures.

Nous savons maintenant comment une application de l'espace utilisateur peut requérir l'aide du système en demandant l'exécution d'un appel système. Notons qu'il existe essentiellement deux types d'appels système : ceux qui offrent un service au processus appelant comme `getpid()`, `fork()`, `sigaction()`, `brk()`, etc. et ceux qui implémentent des méthodes associées à un descripteur comme `open()`, `close()`, `read()`, `write()`, `ioctl()`, `select()`, `mmap()`... Bien sûr, il s'agit d'un descripteur pouvant représenter un fichier classique, mais également une *socket*, un périphérique caractère ou bloc, un tube de communication, etc.

2.2 Communications avec l'espace utilisateur

L'espace d'adressage virtuel d'un processus s'étend classiquement de `0000 0000` à `BFFF FFFF` sur un processeur 32 bits et de `0000 0000 0000 0000` à `0000 7FFF FFFF FFFF` sur un processeur 64 bits. À l'intérieur de cet espace, la MMU (*Memory Management Unit*) projette des pages de mémoire en assurant la conversion vers les adresses physiques.

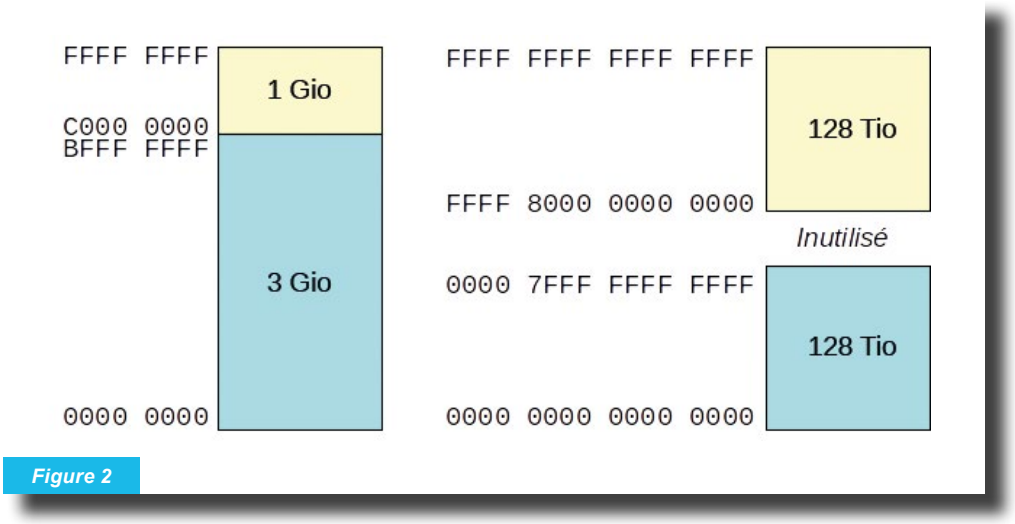


Figure 2

Espaces d'adressage.

L'espace réservé au noyau se situe généralement entre `C000 0000` et `FFFF FFFF` sur une architecture 32 bits, et entre `FFFF 8000 0000 0000` à `FFFF FFFF FFFF FFFF` sur 64 bits (on peut remarquer que les processeurs 64 bits actuels ne gèrent « que » 256 To sur les 16 Eo qu'ils pourraient couvrir en théorie) (voir figure 2).

Un processus ne peut jamais accéder à la mémoire du kernel, car la MMU protège les adresses de ce dernier en les marquant comme accessibles en mode superviseur seulement. À l'inverse, pour des raisons de portabilité, un driver se trouvant dans le noyau **ne doit jamais accéder directement** à la mémoire d'un processus !

Pour échanger des données entre appel système et espace utilisateur, deux possibilités s'offrent à nous :

- une **copie** des données avec les fonctions spécialisées `copy_to_user()` et `copy_from_user()` comme on le fait traditionnellement dans les appels système `read()`, `write()`, `ioctl()` par exemple ;
- une **projection** directe des pages de mémoire du noyau dans l'espace d'adressage du processus appelant à l'aide de la fonction `remap_pfn_range()` ainsi que l'utilise l'appel système `mmap()`.

Voici un exemple de code noyau qui propose un appel-système `read()` pour l'espace utilisateur. Ce module extrêmement simple installe un simili-driver de type caractère, dont le fichier spécial apparaît automatiquement sous le nom `/dev/exemple-01`. Une lecture de ce fichier nous renvoie un message de salutation personnalisé (incluant le *PID* et le nom de la commande appelante).

Fichier

exemple-01.c :

```
#include <asm/uaccess.h>
#include <linux/fs.h>
#include <linux/miscdevice.h>
#include <linux/module.h>
#include <linux/sched.h>

static ssize_t ex_read(struct file *filp, char *u_buffer,
                       size_t max_lg, loff_t *offset)
{
  int lg;
  char k_msg[128];

 // Remplir une chaîne de salutation avec le PID de l'appelant.
  snprintf(k_msg, 128, "Hello '%s/%u'!\n", current->comm, current->pid);

 // Calculer la longueur restant à renvoyer.
  lg = strlen(k_msg) - (*offset);
  if (lg <= 0)
    return 0;
 // Tronquer si nécessaire (si le buffer fourni est trop petit).
  if (lg > max_lg)
    lg = max_lg;
 // Copier le message dans le buffer de l'espace utilisateur.
  if (copy_to_user(u_buffer, &k_msg[*offset], lg) != 0)
    return -EFAULT;
  *offset += lg;
  return lg;
}

  static struct file_operations ex_fops = {
    .owner    =  THIS_MODULE,
    .read     =  ex_read,
  };

  static struct miscdevice ex_misc = {
        .minor          = MISC_DYNAMIC_MINOR,
        .name           = THIS_MODULE->name,
        .fops           = & ex_fops,
  };

static int __init ex_init (void)
{
 // Initialiser un driver caractère de classe Misc.
  return misc_register(& ex_misc);
}

static void __exit ex_exit (void)
{
  misc_deregister(& ex_misc);
}

  module_init(ex_init);
  module_exit(ex_exit);
[...]
```

Après chargement du module, on peut consulter avec la commande **cat** le fichier spécial pour voir notre message :

Terminal

```
[HS]$ sudo insmod exemple-01.ko
[HS]$ sudo cat /dev/exemple_01
Hello 'cat/16682'!
[HS]$ sudo cat /dev/exemple_01
Hello 'cat/16684'!
[HS]$ sudo rmmod exemple_01
```

L'inconvénient de ce mécanisme est de nécessiter une copie des données. Ceci prend du temps et est donc mal adapté aux communications avec un gros débit de données (flux vidéo par exemple). Dans ce cas on préfère projeter dans l'espace utilisateur des pages mémoire appartenant au driver. Ceci s'obtient avec la fonction **`remap_pfn_range()`** qui modifie la configuration de la MMU pour le processus appelant.

L'exemple suivant permet à un processus de projeter dans sa mémoire une page sur laquelle le kernel vient écrire toutes les secondes un message (le nombre de secondes écoulées depuis le 1er janvier 1970 et le complément en microsecondes).

Les exemples que je développe ici sont écrits en essayant de les rendre lisibles et compréhensibles. Ils ne suivent pas toujours les recommandations d'écriture du kernel (j'utilise plusieurs variables globales non-indispensables par exemple) et ne sont pas optimisés.

Fichier

exemple-02.c :

```
[...]
  static char * ex_msg_string = NULL;
  static struct timer_list ex_timer;

static int ex_mmap (struct file * filp, struct vm_area_struct * vma)
{
 // Vérifier que la projection demandée ne soit pas trop grande.
  if ((unsigned long) (vma->vm_end - vma->vm_start) > PAGE_SIZE)
    return -EINVAL;
 // Réaliser la nouvelle projection
 return remap_pfn_range(vma,
                        (unsigned long) (vma->vm_start),
                        virt_to_phys(ex_msg_string) >> PAGE_SHIFT,
                        vma->vm_end - vma->vm_start,
                        vma->vm_page_prot);
}

static void ex_timer_function (unsigned long arg)
{
  struct timeval tv;

  // Lire l'heure et l'inscrire dans la page partagée.
  do_gettimeofday(& tv);
  snprintf(ex_msg_string, PAGE_SIZE, "\rTime: %ld.%06ld  ",
           tv.tv_sec, tv.tv_usec);
  // Reprogrammer le timer dans une seconde.
  mod_timer(& ex_timer, jiffies + HZ);
}

  static struct file_operations ex_fops = {
    .owner    =  THIS_MODULE,
    .mmap     =  ex_mmap,
  };
```

```
  static struct miscdevice ex_misc = {
        .minor          = MISC_DYNAMIC_MINOR,
        .name           = THIS_MODULE->name,
        .fops           = & ex_fops,
  };

static int __init ex_init (void)
{
  int err;

  ex_msg_string = kzalloc(PAGE_SIZE, GFP_KERNEL);
  if (! ex_msg_string)
    return -ENOMEM;
  // Le buffer est réservé par le noyau même s'il sera partagé
  // avec le processus appelant.
  SetPageReserved(virt_to_page(ex_msg_string));

 // Initialiser un timer à la seconde pour modifier le buffer.
  init_timer(& ex_timer);
  ex_timer.function = ex_timer_function;
  ex_timer.expires = jiffies + HZ;
  add_timer(& ex_timer);

 // Initialiser un driver caractère de classe Misc
  err = misc_register(& ex_misc);
  if (err != 0) {
    ClearPageReserved(virt_to_page(ex_msg_string));
    kfree(ex_msg_string);
  }
  return err;
}

static void __exit ex_exit (void)
{
  del_timer(& ex_timer);
  misc_deregister(& ex_misc);
  ClearPageReserved(virt_to_page(ex_msg_string));
  kfree(ex_msg_string);
}

[...]
```

Pour accéder au contenu de notre driver, il faut un processus invoquant l'appel système **mmap()**. Voici un petit programme qui réclame la projection et en affiche le contenu tous les dixièmes de secondes.

Fichier

mmap.c :

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

int main(int argc, char *argv[])
{
  int fd;
  char *ptr;
```

```
    if (argc != 2) {
      fprintf(stderr, "usage: %s <device>\n", argv[0]);
      exit(1);
    }
    if ((fd = open(argv[1], O_RDONLY, 0)) < 0) {
      perror(argv[1]);
      exit(1);
    }
    ptr = mmap(NULL, 32, PROT_READ, MAP_SHARED, fd, 0);
    if (ptr == MAP_FAILED) {
      perror("mmap");
      exit(1);
    }
    while (1) {
      fprintf(stderr, "%s", ptr);
      usleep(100000);
    }
    return 0;
  }
```

Lorsqu'on charge le module et qu'on lance le processus, on voit l'heure évoluer toutes les secondes, nous permettant ainsi de vérifier que cette page est bien partagée entre l'espace utilisateur et celui du noyau.

Terminal

```
[HS]$ sudo insmod exemple-02.ko
[HS]$ sudo ./mmap /dev/exemple_02
Time: 1473605894.894373    ^C
[HS]$ sudo rmmod exemple_02
```

2.3 Concurrence d'accès et synchronisation

Dans l'exemple précédent, nous avons deux accès simultanés à la même page mémoire. Ceci est fortement déconseillé, car des incohérences peuvent se produire si deux écritures se produisent en même temps ou si une écriture modifie l'ensemble des données alors qu'une lecture est en cours. Il existe plusieurs moyens de synchroniser les opérations, nous allons voir le plus répandu lorsque les deux accès se font depuis des appels système : les **mutex** (*MUTual EXclusion*).

Les *mutex* du kernel sont très proches de ceux que l'on emploie en programmation multi-*threads*. On définit un *mutex* pour protéger un objet des accès concurrents et l'on se discipline ensuite à prendre le *mutex* avec `mutex_lock_interruptible()` avant tout accès et à le rendre ensuite avec `mutex_lock()`. Le mot *interruptible* au moment du verrouillage fait référence au type de sommeil dans lequel tombera le processus appelant si le *mutex* est déjà pris : un sommeil *interruptible* peut se terminer prématurément si un signal survient (par exemple généré par les touches <**Contrôle**> + <C>) auquel cas nous devons quitter notre appel système en renvoyant l'erreur `ERESTARTSYS`.

Dans l'exemple suivant, nous allons construire une sorte de petite *fifo*, une table dans laquelle on peut venir stocker des données avec un appel système `write()` puis les extraire avec un appel `read()`.

Fichier

exemple-03.c :

```
[...]
  #define MAX_MESSAGES 8
  static int msg_table[MAX_MESSAGES];
  static int nb_msg = 0;
  DEFINE_MUTEX(msg_mtx);

static ssize_t ex_read(struct file *filp, char *u_buffer,
                           size_t max_lg, loff_t *offset)
{
  int lg;
  char k_msg[128];

 // Verrouiller l'accès à la table de messages.
  if (mutex_lock_interruptible(&msg_mtx) != 0)
    return -ERESTARTSYS;
 // S'il n'y a aucun message, indiquer une fin de fichier.
  if (nb_msg == 0) {
    mutex_unlock(&msg_mtx);
    return 0;
  }
 // Lire le premier message, et décaler les autres.
  sprintf(k_msg, "%d\n", msg_table[0]);
  nb_msg --;
  if (nb_msg > 0)
    memmove(msg_table, &msg_table[1], nb_msg*sizeof(int));
  mutex_unlock(&msg_mtx);
 // Renvoyer le message lu si le buffer fourni est assez grand.
  lg = strlen(k_msg);
  if(lg > max_lg)
    return -ENOMEM;
  if (copy_to_user(u_buffer, k_msg, lg) != 0)
    return -EFAULT;
  return lg;
}

static ssize_t ex_write(struct file *filp, const char *u_buffer,
                           size_t lg, loff_t *offset)
{
  char * k_msg;
  int value;

 // Allouer un buffer local et y copier le message transmis.
  k_msg = kmalloc(lg, GFP_KERNEL);
  if (k_msg < 0)
    return -ENOMEM;
  if (copy_from_user(k_msg, u_buffer, lg) != 0) {
    kfree(k_msg);
    return -EFAULT;
  }
 // Lire une valeur numérique depuis le buffer.
  if (sscanf(k_msg, "%d", & value) != 1) {
    kfree(k_msg);
    return -EINVAL;
  }
  kfree(k_msg);
```

```
// Verrouiller l'accès à la table de message
  if (mutex_lock_interruptible(&msg_mtx) != 0)
    return -ERESTARTSYS;
 // Si la table est pleine, renvoyer une erreur.
  if (nb_msg == MAX_MESSAGES) {
    mutex_unlock(&msg_mtx);
    return -EBUSY;
  }
 //  Ajouter le message et libérer l'accès à la table.
  msg_table[nb_msg] = value;
  nb_msg ++;
  mutex_unlock(&msg_mtx);
  return lg;
}
[...]
```

Notre table est très petite (huit entrées) pour que l'on atteigne facilement la limite en testant le module.

Terminal

```
[HS]$ sudo  insmod  exemple-03.ko
```

Écrivons et relisons une valeur :

Terminal

```
[HS]$ sudo  sh  -c  «echo 11 > /dev/exemple_03"
[HS]$ sudo  cat  /dev/exemple_03
11
```

Si on essaye de lire alors que la table est vide, l'opération se termine immédiatement :

Terminal

```
[HS]$ sudo cat /dev/exemple_03
```

Si on essaye d'écrire alors que la table est pleine, c'est l'erreur *busy* qui se produit :

Terminal

```
[HS]$ for i in $(seq 1 9); do sudo sh -c «echo $i > /dev/exemple_03"; done
sh: ligne 0 : echo: erreur d'écriture : Périphérique ou ressource occupé
[HS]$ sudo cat /dev/exemple_03
1
2
3
4
5
6
7
8
[HS]$ sudo rmmod exemple_03
```

2.4 Sommeil et ordonnancement

Notre exemple précédent marchait bien, mais on peut lui reprocher un détail : une tentative de lecture alors que la file est vide et se termine immédiatement en fin de fichier. De même, une écriture dans une table pleine échoue en erreur. On pourrait préférer que ces opérations restent bloquantes jusqu'à ce qu'elles soient possibles.

Pour cela, on va utiliser une structure de données permettant d'endormir le processus appelant jusqu'à ce qu'on le réveille explicitement : une **waitqueue**. Une *waitqueue* rappelle un peu la variable-condition qu'on peut trouver dans l'API des *threads* POSIX. Lorsque je présente la *waitqueue* lors de mes sessions de formation, je la compare souvent à une cloche, à côté de laquelle il est possible de s'endormir et sur laquelle on peut donner un coup. Si personne ne dort au moment du coup sonné cela n'a pas d'importance, il n'est pas mémorisé (au contraire par exemple d'un sémaphore).

Dans notre module, nous ajouterons deux *waitqueues* : une pour endormir la lecture si la table est vide et une pour endormir l'écriture quand la table est déjà pleine. Symétriquement, c'est l'opération d'écriture qui viendra donner un coup sur la *waitqueue* « table vide » et l'opération de lecture pour la *waitqueue* « table pleine ».

Je ne présente ci-dessous que les portions de code modifiées :

Fichier

exemple-04.c :

```
[...]
  static int msg_table[MAX_MESSAGES];
  static int nb_msg = 0;
  DEFINE_MUTEX(msg_mtx);
  DECLARE_WAIT_QUEUE_HEAD(msg_tbl_full_wq);
  DECLARE_WAIT_QUEUE_HEAD(msg_tbl_empty_wq);

static ssize_t ex_read(struct file *filp, char *u_buffer,
                       size_t max_lg, loff_t *offset)
{
  [...]
 // Verrouiller l'accès à la table de messages.
  if (mutex_lock_interruptible(&msg_mtx) != 0)
    return -ERESTARTSYS;
 // Tant qu'il n'y a aucun message, dormir en attente dans la waitqueue.
  while (nb_msg == 0) {
    mutex_unlock(&msg_mtx);
    err = wait_event_interruptible(msg_tbl_empty_wq, nb_msg!=0);
    if (err != 0)
      return -ERESTARTSYS;
    if (mutex_lock_interruptible(&msg_mtx)!= 0)
      return -ERESTARTSYS;
  }
  [...]
 // Notifier un éventuel écrivain bloqué.
  wake_up_interruptible(&msg_tbl_full_wq);
  [...]
}

static ssize_t ex_write(struct file *filp, const char *u_buffer,
                        size_t lg, loff_t *offset)
{
  [...]
 // Tant que la table est pleine, dormir en attente dans la waitqueue.
  while (nb_msg == MAX_MESSAGES) {
    mutex_unlock(&msg_mtx);
    err = wait_event_interruptible(msg_tbl_full_wq, nb_msg!=MAX_MESSAGES);
    if (err != 0)
      return -ERESTARTSYS;
    if (mutex_lock_interruptible(&msg_mtx)!= 0)
      return -ERESTARTSYS;
  }
  [...]
```

```
    // Notifier un éventuel lecteur bloqué.
     wake_up_interruptible(&msg_tbl_empty_wq);
     [...]
}
[...]
```

Les différences d'exécution avec l'exemple précédent sont surtout dynamiques, il est difficile d'en donner un compte-rendu ici. Précisons que :

- une lecture lorsque la file est vide reste bloquée en attendant une nouvelle valeur (envoyée avec **echo** depuis un autre terminal) ou la pression sur <**Contrôle**> + <**C**> pour arrêter le processus,
- l'écriture de valeurs successives bloque lorsque la table est pleine (au bout de huit valeurs) jusqu'à ce qu'on vienne lire un élément depuis un autre terminal ou que l'on presse <**Contrôle**> + <**C**>.

Ce mécanisme de mise en sommeil et réveil depuis un autre contexte est crucial pour le fonctionnement du système. Examinez le résultat d'une commande **ps aux**, vous verrez des dizaines de processus endormis (état *Sleeping*) dans des *waitqueues*.

3. COMMUNICATION AVEC LE MATÉRIEL

Les exemples que nous avons vus précédemment ne communiquent qu'entre appels système et espace utilisateur. Il n'y a pas de dialogue direct avec le matériel sous-jacent. C'est pourtant le rôle essentiel de la plupart des drivers.

3.1 Entrées/sorties vers les périphériques

Pendant longtemps, la communication avec les périphériques externes au processeur demandait des instructions assembleur spéciales (**inb**, **inw**, **inl**, **outb**, **outw**, **outl**...) qui agissaient sur des ports d'entrées-sorties dans un espace d'adressage totalement distinct de la mémoire. Sur la plupart des processeurs actuels c'est terminé, la MMU est capable de projeter dans l'espace d'adressage du processeur les points d'entrée des périphériques externes (les registres) de manière totalement équivalente à la mémoire classique.

Pour dialoguer avec un matériel, il nous faut d'abord connaître l'adresse physique de ce périphérique. Deux cas se présentent en général :

- soit il est accessible par un bus (par exemple *PCI*) permettant d'énumérer les périphériques connectés et de négocier leurs conditions d'accès, auquel cas l'adresse physique nous est directement transmise dans notre méthode **probe()**, invoquée lors de la détection du matériel ;
- soit l'adresse est indiquée lors de la configuration du matériel (par exemple dans le *BIOS* sur un PC ou dans le *device tree* sur un processeur ARM) et l'on peut la connaître grâce à une fonction d'interrogation comme **platform_get_resource()**.

Connaissant l'adresse physique de notre matériel, nous pouvons demander une projection dans l'espace d'adressage du noyau avec (voir **linux-4.7/include/asm-generic/io.h**) :

Fichier

```
void __iomem *ioremap_nocache(phys_addr_t offset, size_t size);
```

L'adresse physique est passée en premier argument, suivie de la taille à projeter. L'adresse renvoyée se trouve dans l'espace d'adressage virtuel du kernel. On voit que le pointeur est marqué d'un attribut spécifique : **__iomem**. En effet, les registres auxquels on accède avec cette projection doivent être lus et écrits directement. Ceci signifie que l'on doit désactiver le cache du processeur sur ces zones d'adressage, pour éviter que deux lectures successives d'une adresse renvoient la même valeur (celle du cache) alors que l'état indiqué dans le registre du périphérique a changé. En outre, on doit éviter que le compilateur lui-même se mêle de réordonner les accès pour faire des optimisations. Par exemple, il existe des périphériques où l'on doit écrire successivement le poids fort puis le poids faible d'une valeur à la même adresse pour la transmettre complètement. Il serait malvenu que le compilateur supprime la première écriture au prétexte que d'après lui la seconde suffira. Pour cela, on évite les accès en lecture/écriture directs sur la plage de pointeurs renvoyés, en préférant les instructions suivantes qui agissent sur des pointeurs **__iomem** :

Fichier

```
u8   ioread8   (void __iomem *addr);
u16  ioread16  (void __iomem *addr);
u32  ioread32  (void __iomem *addr);
void iowrite8  (u8 value, void __iomem *addr);
void iowrite16 (u16 value, void __iomem *addr);
void iowrite32 (u32 value, void __iomem *addr);
```

Outre l'accès en lecture ou écriture à l'adresse d'entrée-sortie, ces fonctions contiennent des barrières mémoires – instructions fictives qui imposent un point d'arrêt à l'optimiseur du compilateur – assurant ainsi que les registres indiqués seront lus ou écrits aussi souvent que nécessaire.

On notera que la projection d'un périphérique dans la mémoire du kernel peut également s'accompagner d'une re-projection dans l'espace utilisateur par l'intermédiaire du **mmap()** vu plus haut.

3.2 Notification par interruption

Nous avons rapidement évoqué le principe des interruptions plus haut. Lorsqu'un périphérique considère qu'il a une information urgente à transmettre au système, il lève un signal électrique à destination du contrôleur d'interruption (APIC, *Advanced Programmable Interrupt Controller*). Ce dernier envoie une demande d'interruption (IRQ, *Interrupt Request*) au processeur. Celui-ci sauvegarde son contexte de travail (registres, pile, etc.) et se branche sur une adresse dépendant du numéro d'interruption pour y exécuter un gestionnaire (*handler*) avant de reprendre son activité précédente.

Le *handler* de bas-niveau est écrit dans le noyau Linux standard et n'est généralement pas modifié. Nous pouvons néanmoins facilement lui fournir une fonction de plus haut-niveau (ISR, *Interrupt Service Routine*) qu'il invoquera. Si plusieurs ISR sont fournies, elles sont toutes appelées successivement.

La fonction (voir **linux-4.7/include/linux/interrupt.h**) suivante permet d'installer une routine de service sur l'interruption dont le numéro est passé en premier argument :

Fichier

```
int request_irq (unsigned int irq, irq_handler_t handler,
                 unsigned long flags, const char *name, void *id)
```

Notre routine doit avoir la forme :

Fichier

```
irqreturn_t  my_function (int irq, void *id)
```

Elle reçoit en argument le numéro de l'interruption ayant déclenché son invocation (pour le cas où la même routine serve à traiter plusieurs interruptions), suivi du pointeur générique que nous avons fourni en dernier argument de **request_irq()** et qui peut par exemple viser une structure personnalisée contenant les informations personnelles du driver. La fonction doit interroger son matériel pour déterminer si c'est bien lui qui a provoqué l'interruption auquel cas elle renverra **IRQ_HANDLED**. À l'inverse, elle renverra **IRQ_NONE** si l'interruption a été déclenchée par un autre périphérique dont le driver sera invoqué par la suite.

Pendant le déroulement d'une routine de service d'interruption, on se trouve dans un contexte particulier qui empêche certaines opérations : pas d'accès à l'espace utilisateur (aucun processus ne nous a appelé) et surtout pas de mise en sommeil (il n'y a pas de processus à endormir). Impossible donc d'utiliser des *mutex*.

On rencontre toutefois très souvent un schéma où la routine d'interruption reçoit des données (provenant par exemple d'un périphérique d'acquisition) et les stocke dans un *buffer* en attendant qu'un appel système **read()** vienne les collecter et les envoyer à l'applicatif se trouvant dans l'espace utilisateur. Il est donc indispensable de disposer d'un moyen d'empêcher les accès concurrents au *buffer* et aux variables servant à gérer celui-ci. Ce mécanisme existe et se nomme le **spinlock**. Assez similaire dans son usage au *mutex*, le *spinlock* assure une attente active sans sommeil. Ainsi les fonctions **spin_lock()** et **spin_unlock()** peuvent être invoquées depuis une routine d'interruption.

Dans un appel système, ce sont les fonctions **spin_lock_irqsave()** et **spin_unlock_irqrestore()** que l'on appellera afin de prendre le *spinlock* et de couper les interruptions sur le processeur appelant puis de restaurer les interruptions en relâchant le *spinlock*. Bien que ce mécanisme paraisse bizarre au premier abord, il faut savoir que c'est un moyen parfaitement adapté pour synchroniser appels système et routine d'interruption de manière parfaitement portable, que l'architecture soit uni-cœur ou multi-cœur.

Pour voir un exemple très simple de *handler* d'interruption, je reprends le principe de l'exemple précédent en le modifiant comme suit :

- la table ne contient plus des nombres, mais des horodatages en secondes depuis le 01/01/1970 et microsecondes ;
- l'appel système **write()** a disparu ainsi que la *waitqueue* « table pleine » ;
- un *handler* d'interruption viendra ajouter son horodatage à chaque déclenchement et l'appel système **read()** renverra les horodatages vers l'espace utilisateur ;
- le *mutex* a été remplacé par un *spinlock* :

Fichier

exemple-05.c :

```
[...]
  #define MAX_TV 8
  static struct timeval tv_table[MAX_TV];
  static int nb_tv = 0;
  static spinlock_t tv_spl;
  DECLARE_WAIT_QUEUE_HEAD(tv_tbl_empty_wq);
 // Le numero d'interruption est un paramètre du module.
  static int tv_irq = 10;
  module_param_named(irq, tv_irq, int, 0600);

static ssize_t ex_read(struct file *filp, char *u_buffer,
size_t max_lg, loff_t *offset)
{
  unsigned long irqs;
  int   err;
  char  k_msg[128];
  int   lg;

 // Verrouiller l'accès à la table des horodatages.
  spin_lock_irqsave(&tv_spl, irqs);
 // Tant qu'il n'y a aucun horodatage, dormir en attente dans la waitqueue.
  while (nb_tv == 0) {
    spin_unlock_irqrestore(&tv_spl, irqs);
    err = wait_event_interruptible(tv_tbl_empty_wq, nb_tv!=0);
    if (err != 0)
      return -ERESTARTSYS;
    spin_lock_irqsave(&tv_spl, irqs);
  }
 // Lire le premier message, et décaler les autres.
  sprintf(k_msg, "%ld.%06ld\n", tv_table[0].tv_sec, tv_table[0].tv_usec);
  nb_tv --;
  if (nb_tv > 0)
    memmove(tv_table, &tv_table[1], nb_tv*sizeof(struct timeval));
  spin_unlock_irqrestore(&tv_spl, irqs);
 // Renvoyer le message lu si le buffer fourni est assez grand.
  [...]
}

static irqreturn_t ex_irq_handler(int irq, void * id)
{
  spin_lock(&tv_spl);
  if (nb_tv < MAX_TV) {
    do_gettimeofday(&(tv_table[nb_tv]));
    nb_tv ++;
  }
  spin_unlock(&tv_spl);
 // Notifier un éventuel lecteur bloqué.
  wake_up_interruptible(&tv_tbl_empty_wq);
  return IRQ_HANDLED;
}

static int __init ex_init (void)
{
  int err;
  spin_lock_init(&tv_spl);
  err = request_irq(tv_irq, ex_irq_handler, IRQF_SHARED,  THIS_MODULE->name,
THIS_MODULE->name);
  if (err != 0)
    return err;
  return misc_register(& ex_misc);
}

[...]
```

Au chargement de ce module, on peut préciser le numéro d'interruption à horodater. Je propose de se placer sur l'interruption « souris » du système. Pour déterminer celle-ci, le plus simple est d'exécuter :

Terminal

```
[HS]$ watch -n 0,1 cat /proc/interrupts
```

On observe quel numéro évolue lorsque l'on clique (par exemple `17` sur mon poste). Le numéro est passé en argument sur la ligne de `insmod` :

Terminal

```
[HS]$ sudo  insmod  exemple-05.ko  irq=17
[HS]$ sudo cat /dev/exemple_05
1473635000.673606
1473635001.680602
1473635004.715601
1473635005.598596
1473635005.686594
1473635005.918591
1473635006.006595
^C
[HS]$ sudo  rmmod  exemple-05.ko
[HS]$
```

CONCLUSION

Cet article est une courte introduction à la programmation noyau, et nous n'avons fait qu'effleurer les mécanismes mis en œuvre pour les entrées/sorties et le traitement des interruptions. Nous n'avons pas parlé des traitements différés (*bottom halves* et *threaded interrupt*) d'interruptions, des autres appels système que `read()`, `write()` et `mmap()` (par exemple `ioctl()`, `select()`, etc.), non plus que de la détection des périphériques (méthodes `probe()` et `disconnect()` du driver) ni de la gestion de la mémoire du kernel (`kmalloc()`, `vmalloc()`, `kmap()`, etc.). J'espère avoir néanmoins fourni une petite boîte à outils permettant de lire assez facilement le code source des drivers du kernel, et de mieux comprendre les interactions entre les applications et le noyau. ■

POUR ALLER PLUS LOIN

Je conseille la lecture régulière du site LWN (*Linux Weekly News*), et plus particulièrement sa rubrique Kernel (**https://lwn.net/Kernel**). La consultation de la dernière édition est réservée aux abonnés, mais les précédentes sont lisibles par tous. Naturellement, je vous encourage à vous abonner pour soutenir l'excellent travail de documentation de ce site.

Pour apprendre à produire du code de bonne qualité pour le noyau Linux, il existe le projet Eudyptula Challenge (**http://eudyptula-challenge.org/**) : une fois inscrit – gratuitement – vous recevrez un exercice par mail, auquel vous devrez répondre pour passer au niveau suivant, etc. La qualité des exercices est excellente, mais ce projet est un peu victime de son succès, et il faut parfois attendre plusieurs semaines pour avoir la correction de notre réponse.

CAS N°2

CLOISONNEMENT DES PROCESSUS AU SEIN DU NOYAU LINUX

Sylvain NAYROLLES

La technologie des conteneurs, comme LXC ou Docker, repose sur des fonctionnalités du noyau permettant de réaliser le cloisonnement des ressources d'un processus.

Les conteneurs reposent essentiellement sur deux fonctionnalités du noyau : les **namespaces** et les **controls groups** ou **cgroups**. Le développement des namespaces fut initié en 2002 dans le noyau, portant essentiellement sur l'isolation des systèmes de fichiers, mais ils permettent actuellement d'affiner encore plus l'isolation des processus entre eux. Les cgroups quant à eux, ont comme origine la volonté de deux ingénieurs de chez **Google**, Paul Menage et Rohit Seth, d'offrir au noyau des fonctionnalités de conteneur. C'est d'ailleurs avec ce terme que ces ingénieurs avaient décrit leur évolution au tout début. Afin de ne pas créer de confusion, justement avec les namespaces, il était préférable de changer le nom pour « control groups » ou encore cgroups. Ils permettent de contrôler les ressources d'un processus, telles que la consommation mémoire, ou encore le nombre de CPU.

1. NAMESPACE

Les namespaces permettent de limiter les interactions entre processus. Plus précisément, ils permettent de regrouper un ensemble de processus selon un ou plusieurs critères. Dans les versions récentes du noyau, il est possible d'isoler les processus selon cinq critères principaux :

- **mount** points de montage ;
- **IPC** System V IPC, file de messages POSIX ;
- **UTS Hostname** et nom de domaine NIS ;
- **network** interface, pile réseau, ports, routes, etc. ;
- **pid** processus ID ;
- **user** ID utilisateur ou ID de group.

La manipulation des namespaces se fait via trois appels systèmes :

- `clone` ; prend en paramètre un ensemble de *flags* permettant au nouveau processus d'être créé dans un ensemble de nouveaux namespace ;
- `setns` ; permet de rejoindre un namespace existant en précisant `fd` ouvert sur un fichier présent dans le répertoire `/proc/[pid]/ns` ;
- `unshare` ; permet de créer un nouveau processus en détachant un ou plusieurs namespaces de son parent.

Chaque namespace a sa particularité, plus ou moins complexe à appréhender. Ces appels systèmes sont donc tous disponibles via la **GLibC**, mais il existe des utilitaires **bash** permettant de jouer un peu avec ces derniers. Le principal outil est la commande `unshare` directement inspirée de l'appel système associé. Ce dernier permet de lancer un processus en le détachant des namespaces de son parent :

Terminal

```
$ sudo unshare -p -f /bin/sh -c "echo $$"
1
```

La variable `$$` contient le pid courant. Le résultat est bien `1`, c'est-à-dire le pid du processus d'init. Nous reviendrons un peu plus tard sur l'ensemble et la pertinence de cette commande.

1.1 User namespace

L'user namespace permet d'isoler les id utilisateur, les id de group, le répertoire root ou encore les capacités. Le user namespace est donc un peu particulier par rapport aux autres, car il permet d'isoler des attributs servant au contrôle d'accès de ressources, pouvant être gérés par d'autres namespaces. Chaque non user namespace doit donc référer un user namespace lors de sa création. Quand plusieurs namespaces sont créés en même temps via la commande `clone`, il est assuré que le user namespace sera le premier créé, permettant aux autres de le référer.

Ceci permet de créer des processus privilégiés, initiés par l'utilisateur `0`, localement au user namespace. Il est aussi possible dans les versions récentes du noyau de créer des règles de *mapping* entre les utilisateurs de deux namespaces hiérarchiques. Il est donc possible de *mapper* un utilisateur non privilégié dans le user namespace parent avec l'utilisateur privilégié, id `0`, dans le namespace enfant. C'est ce que propose l'option `--map-root-user` ou `-r` de `unshare`.

Terminal

```
# unshare --map-root-user /bin/bash
# id
uid=0(root) gid=0(root) groups=0(root),65534(nogroup)
# more /proc/self/uid_map
         0       1000          1
```

Ici nous observons que dans le namespace courant nous sommes bien l'utilisateur privilégié, mais que nous avons utilisé la table de *mapping*, disponible via `/proc/self/uid_map`, qui nous fait correspondre à l'utilisateur `1000` dans le namespace parent.

La gestion de droit dans le user namespace peut, à lui seul, faire l'objet d'un article entier. C'est pour cela que je vous invite à faire quelques recherches, car tout n'est pas trivial, dû à la position transverse du namespace user par rapport aux autres. Par exemple, il n'est pas possible de monter n'importe quel type de système de fichiers dans un mount namespace attaché à un user namespace non privilégié. On considère le mount namespace non privilégié.
De plus, certaines capacités ne sont pas disponibles dans un user namespace non privilégié tel que `CAP_SYS_MODULE`, permettant de charger un module noyau. Cela s'explique par le fait que ce dernier, une fois dans le noyau, s'abstrait des namespaces des processus utilisateurs.

1.2 PID namespace

Le PID namespace permet de cloisonner l'ensemble d'ID des processus, et permet d'attribuer des ID différents entre le namespace parent et le namespace enfant.
Mais l'arbre des processus continue bel et bien d'exister. Le processus sera donc visible de tous les PID namespace parents auxquels il appartient. Ceci constitue un *mapping* des IDs et ne permet pas de cacher des processus aux parents. Le processus ainsi créé aura un PID dans chaque PID namespace auquel il appartient.

Le processus créant le namespace, initialise le compteur pour les nouveaux ID. Nous allons par la suite distinguer le processus créant le namespace et le premier processus créé dans ce dernier. Le premier garantit l'existence du namespace jusqu'à sa mort ; le second garantit l'existence de l'arbre des processus dans ce namespace et aura donc le PID `1`, soit celui du processus d'init. Tous les processus zombies appartenant au PID namespace seront donc rattachés à ce dernier. Mais si ce dernier se termine, un message `SIGKILL` sera envoyé à tous les enfants. Il ne sera plus possible de créer de processus par la suite, car le processus d'init sera terminé.

Nous allons créer un nouveau pid namespace :

Terminal
```
# unshare -r -p /bin/bash
bash: fork: Cannot allocate memory
```

Le paramètre `-r` nous permet de créer un user namespace avec un processus non privilégié, en *mappant* l'ID de l'utilisateur courant avec l'ID `0` dans le nouveau namespace. Le paramètre `-p` permet de créer un processus avec `clone` en détachant le PID namespace du parent. Dans cet état, nous n'avons pas encore de process d'init. C'est pour cela que nous avons un message d'erreur (pas très explicite). La commande `unshare` permet donc de lancer la commande en paramètre, après un fork, ce qui aura pour effet de lui attribuer le PID `1` :

Terminal
```
# unshare -r -p -f /bin/bash
# echo $$
1
```

Par contre si nous faisons un `ps -aux` afin de lister les processus actifs, étrangement nous allons voir l'ensemble des processus dans le namespace parent. Ceci s'explique parce qu'un logiciel tel que `ps` se base sur le point de montage `/proc` afin de réaliser ses opérations. Pour résoudre ce problème, il va falloir exécuter le processus dans un nouveau mount namespace et recréer le point de montage `/proc`. `unshare` permet de faire ceci via l'option `--mount-proc` :

Terminal
```
# unshare -p -r -f --mount-proc /bin/bash
# ps -aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.6  0.0  27244  5680 pts/39   S    21:20   0:00 /bin/bash
root         11  0.0  0.0  22680  2628 pts/39   R+   21:20   0:00 ps -aux
```

1.3 Mount namespace

Le mount namespace permet donc d'isoler les points de montage entre les processus. Un processus hérite de l'ensemble des points de montage de son processus parent. Mais une fois dans son namespace, il peut soit en créer, soit en démonter sans affecter ceux de son parent. Nous sommes bien en train de dire qu'un processus peut cacher un ensemble de points de montage, même à son processus parent.

Dans cet exemple, nous allons créer un point de montage de type **tmpfs**, mais qui sera caché des autres processus, y compris celui du parent.

Dans un premier temps, nous allons créer un processus via la commande `unshare` en lui précisant de se détacher du namespace mount de son parent :

Terminal
```
# unshare -m -r /bin/bash
```

L'option `-r` permet de mapper l'utilisateur courant en tant qu'utilisateur avec l'id `0`, c'est-à-dire root dans le nouveau user namespace qui sera donc automatiquement créé. Nous y reviendrons plus tard dans cet article. Ceci va nous permettre de créer un point de montage en root et non pas en user.

Terminal
```
# mkdir /tmp/process_partition
# mount -n -o size=1m -t tmpfs tmpfs /tmp/process_partition
```

Nous allons maintenant créer un fichier dans cette partition :

Terminal
```
# touch /tmp/process_partition/hidden_file
```

Dans notre nouveau namespace, nous pouvons voir notre dossier, notre point de montage ainsi que notre nouveau fichier :

Terminal
```
# df -h
tmpfs           1,0M     0  1,0M   0% /tmp/process_partition
# ls -al /tmp/process_partition
drwxrwxrwt  2 root   root      60 août  28 14:57 .
drwxrwxrwt 19 nobody nogroup 4096 août  28 15:16 ..
-rw-rw-r--  1 root   root       0 août  28 14:57 hidden_file
```

Si maintenant, nous ouvrons un nouveau terminal, dans le user et mount namespace principal puis que nous essayons d'atteindre le fichier dans le point de montage, ceci est impossible, car l'OS ne connaît pas le point de montage. Le fichier est donc rendu inaccessible en dehors d'un dump de la mémoire :

Terminal
```
# df -h
...
```

Le répertoire `/tmp/process_partition` sera créé avec les droits de l'utilisateur du processus parent.

Terminal
```
# ls -aild /tmp/process_partition
drwx------  2 sylvain sylvain   4096 août  28 14:56 /tmp/process_partition
```

Nous venons de voir, que grâce au namespace, il est donc possible de cacher un ensemble de fichiers au processus parent, y compris à l'utilisateur root ! Mais ce comportement est nominal et se justifie par la possibilité d'avoir un processus init dans un namespace qui n'est pas un processus privilégié dans le namespace root. Mais pour cela il nous faut une version du noyau récente, supérieure à 3.9.

Nous commençons à apercevoir la puissance des namespaces, ainsi que les effets qu'ils peuvent avoir sur la configuration de notre système d'exploitation.

1.4 UTS namespace

L'UTS namespace permet de cloisonner deux identifiants que sont le hostname et le nom de domaine NIS. Ce namespace fut introduit avec l'avènement des conteneurs embarquant de plus en plus de services web.

Terminal
```
# unshare -r -u /bin/bash
# hostname
sylvain
# hostname linux-mag
# hostname
linux-mag
```

Nous pouvons constater que le `hostname` ne change que dans le namespace.

1.5 IPC namespace

Les IPC namespace permettent de cloisonner les appels inter processus à un sous-ensemble de processus appartenant au namespace. Il sera donc possible de limiter, par exemple, des mémoires partagées à un sous-ensemble de processus. Ceci dans un souci de cloisonnement avancé entre processus.

1.6 Network namespace

Les network namespace permettent d'isoler les fonctionnalités réseau. Ceci comprend bien sûr la stack IP, mais aussi les interfaces, les routes, ainsi que les règles netfilter ou iptables. Nous comprendrons l'intérêt dans le cadre des conteneurs tels que nous les connaissons, car souvent utilisés dans le monde réseau.

L'utilitaire **`ip`** permet, en plus de l'utilitaire **`unshare`**, de manipuler les network namespace. Il peut par exemple créer un nouveau namespace :

Terminal
```
$ sudo ip netns add <mon_namespace>
```

Le supprimer :

Terminal
```
$ sudo ip netns delete <mon_namespace>
```

Ou bien encore lister les namespaces existants :

Terminal
```
$ sudo ip netns list
```

Enfin, il est possible de lancer une commande dans le network namespace via :

Terminal
```
$ sudo ip netns exec <mon_namespace> <cmd>
```

L'ensemble des namespaces ainsi créés est répertorié dans le répertoire **`/var/run/netns/<mon_namespace>`**. Ce fichier peut être utilisé par un processus via la fonction **`setns`** avec comme argument un *file descriptor* ouvert sur un de ces fichiers.

2. CONTROL GROUPS (CGROUPS)

Les cgroups permettent de contrôler l'utilisation des ressources d'un processus selon plusieurs critères.

Il existe trois méthodes afin de manager ces derniers :

- une API système permettant de manipuler de façon programmatique les cgroups ;
- un système de fichiers, souvent monté au point **`/sys/fs/cgroup`**, offrant donc via une simple API en mode fichier l'ensemble des fonctionnalités ;
- enfin un paquet, **`libcgroup`**, regroupant un ensemble d'utilitaires permettant la création, la suppression, la configuration ainsi que la persistance.

Les cgroups, comme les namespaces, se divisent en sous-systèmes, chacun permettant de gérer un ensemble de ressources :

- **blkio** : permet de contrôler l'accès aux ressources input/output en mode *block* tels que les disques durs ;
- **cpuset** : permet de contrôler l'accès au nombre de cœurs du processeur ;
- **cpuacct** : permet de générer des rapports automatiques sur la consommation CPU ;
- **cpu** : permet de répartir la charge processeur entre les mêmes membres d'un groupe ;
- **devices** : permet de contrôler l'accès à certaines ressources physiques ;
- **freezer** : permet de suspendre ou reprendre des tâches du groupe ;
- **memory** : permet de contrôler l'utilisation de la mémoire pour un ensemble de processus ;
- **net_cls** : permet de tagger les paquets réseau d'un ensemble de processus qui pourront ensuite être manipulés par `tc`, le trafic contrôleur sous linux ;
- **net_prio** : permet de contrôler la priorité du trafic réseau par interface ;
- **ns** : les namespaces.

Dans la suite de cet article, nous allons manipuler les cgroups via les commandes offertes par la libcgroup :

Terminal
```
$ sudo apt-get install cgroup-tools
```

Ceci vous installera un ensemble d'utilitaires qui vous faciliteront la gestion des cgroups dont :

- `lscgroup` permettant de lister les cgroups ;
- `cgm` qui va vous permettre de manager les cgroups ;
- `cgcreate` permettant de créer un cgroup.

Pour créer un nouveau cgroup, il suffit de faire :

Terminal
```
$ sudo cgcreate -t sylvain -a sylvain -g memory:test-cgroup/foo
```

Le paramètre `-t` permet de spécifier l'utilisateur des *tasks* que le *group* contrôlera. Le paramètre `-a` permet de spécifier l'utilisateur qui pourra modifier les paramètres du *group*. Et le paramètre `-g` permet de spécifier un ensemble de sous-systèmes séparés par des virgules ainsi que le nom du nouveau groupe.

Il est possible de définir la valeur max de la mémoire que les membres d'un groupe peuvent utiliser :

Terminal
```
# echo 100000 > /sys/fs/cgroup/memory/test-cgroup/foo/memory.limit_in_bytes
```

Puis nous pouvons exécuter un processus dans ce groupe :

Terminal
```
# cgexec -g memory:test-cgroup/foo python
```

Et enfin, nous pouvons *monitorer* l'usage de la mémoire via :

Terminal
```
# watch cat /sys/fs/cgroup/memory/test-cgroup/foo/memory.usage_in_bytes
```

Il existe de nombreux paramètres par sous-systèmes. Nous allons ici détailler certains paramètres des sous-systèmes memory, cpuset et freezer afin de vous sensibiliser avec la puissance des cgroups.

2.1 Memory

Voici les paramètres intéressants du sous-système memory :

`memory.usage_in_bytes`	Permet d'afficher la mémoire utilisée par les processus du groupe
`memory.memsw.usage_in_bytes`	Permet d'afficher le swap utilisé par les processus du groupe
`memory.limit_in_bytes`	Permet de définir une limite d'utilisation de la mémoire
`memory.memsw.limit_in_bytes`	Permet de définir une limite d'utilisation du swap
`memory.kmem.limit_in_bytes`	Permet de définir une limite d'utilisation de la mémoire du noyau
`memory.kmem.usage_in_bytes`	Permet d'afficher la mémoire utilisée par le noyau pour les processus

Pour plus de paramètres, on peut se référer à la documentation kernel sur les cgroup memory [1].

2.2 CPUset

Voici un extrait des paramètres pour gérer les cpu par groupe :

`cpuset.cpus`	Permet de définir l'ensemble des cpu utilisés par les processus
`cpuset.cpu_exclusive`	Permet de définir l'exclusivité de l'usage de ces cpu

Pour plus de paramètres, on peut se référer à la documentation kernel sur les cgroup cpuset [2].

2.3 Freezer

Ce sous-système est relativement simple et ne comporte que très peu de paramètres. Le plus important est le paramètre `freezer.state` permettant de définir l'état des processus du groupe :

- ⇨ `THAWED` : les tâches sont ordonnancées par le noyau sans restriction ;
- ⇨ `FREEZING` : cette valeur est en lecture seule et permet de connaître l'état transitoire ;
- ⇨ `FROZEN` : toutes les tâches sont suspendues.

CONCLUSION

Au travers de cet article, nous avons pu découvrir les coulisses des technologies telles que **Docker**, ou bien **LXC**. Nous avons pu voir qu'il était donc possible de contrôler avec justesse les paramètres noyau d'un ensemble de processus. Pour aller plus loin, il aurait été intéressant de parler d'**AppArmor**, qui permet d'ajouter une dimension sécuritaire aux notions de namespace et cgroup.

Mais ces notions peuvent aussi être intéressantes, comme pour tester le comportement d'un logiciel quand ses ressources sont limitées, ou bien encore quand l'on veut cloisonner ses sous-processus, par exemple dans un serveur web. ▉

RÉFÉRENCES

[1] Documentation du cgroup memory : **https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt**

[2] Documentation du cgroup cpuset : **https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt**

2 EXPÉRIMENTEZ

CAS N°3

COMMENT DÉBOGUER LE NOYAU ?

Sylvain NAYROLLES

Le débogage noyau a longtemps été rendu difficile par le manque d'outils. Aujourd'hui, cette tâche est rendue facile par l'intégration d'outils dans la mainstream du noyau, ainsi que par des outils de virtualisation.

Le débogage noyau fut longtemps rendu difficile par l'absence d'outils, et surtout par la volonté de Linus Torvald de ne pas en fournir. Ce dernier le justifiait par la volonté de ne pas faciliter le développement du noyau, ayant peur d'une baisse de la qualité du code produit. Depuis, des modules tels que **Kgdb** furent introduits dans la *mainstream*. Puis, avec l'avènement des outils de virtualisation, d'autres vecteurs de débogage ont donc émergé, tels que la combinaison de **gdb** et **qemu**. Nous allons donc faire un petit tour d'horizon des différentes techniques de débogage du noyau.

1. GDB

Gdb est le débogueur des développeurs sous Linux. Faisant partie de la suite des GNU tools, il se décline aussi sur la grande majorité des architectures présentes sur le marché.

Il comprend les symboles de débogage au format DWARF3, les mêmes que ceux générés par ses amis de toujours `cc` et `gcc`.

1.1 Ligne de commandes

Gdb possède une interface CLI très évoluée. Il permet aussi de faire du débogage à distance, et c'est cette fonctionnalité qui va nous intéresser.

Pour lancer le débogage, il suffit d'exécuter gdb :

Terminal
```
$ gdb <binaire_du_programme>
(gdb)
```

Pour lancer une session à distance, il suffit de le préciser via la commande :

Terminal
```
(gdb) target remote <addresse>
```

Par défaut, `gdb` cherche la fonction `main`, et s'en sert comme premier point d'arrêt. Dans le cadre du kernel, il n'y a pas de fonction `main`, mais nous verrons que le noyau nous offre d'autres possibilités.

Ensuite, il est possible de manipuler l'exécution via les commandes suivantes :

`b`/`break`	Permet de définir un point d'arrêt par adresses ou par symboles
`i`/`info` ⇨ `args` ⇨ `registers`	Affiche des informations concernant les arguments du programme, ou bien l'adresse des breakpoints installés, ou encore l'état des registres du processeur
`s`/`step`	Pas à pas qui entre dans la fonction
`n`/`next`	Pas à pas qui ne rentre pas dans la fonction
`bt`/`backtrace`	Affiche la pile d'appel
`x`	Examine la mémoire
`c`	Continue
`run`	Lance le processus de débogage
`layout asm`	Permet d'afficher le code désassemblé et le code source en parallèle

2. KGDB

Kgdb fut développé avec la volonté de fournir un vrai outil d'aide pour les développeurs de modules ou drivers du noyau. Il fut longtemps livré sous forme de patch, car Linus Torvald en personne lui refusait la légitimité d'intégrer la *mainstream*. Pour se justifier, il mettait en avant qu'un tel outil pouvait faciliter, justement, l'accès au développement noyau, et donc par conséquent en diminuerait la qualité. Ce jugement vaut ce qu'il vaut, mais Kgdb fût introduit dans la *mainstream* à partir de mai 2008, suite à un travail colossal de Ingor Molnar visant à limiter l'impact sur le code existant.

Pour activer Kgdb, il suffit donc de l'activer lors de la compilation du noyau (voir figure 1).

```
-*- KGDB: kernel debugger
<*>   KGDB: use kgdb over the serial console
[ ]   KGDB: internal test suite
[*]   KGDB: Allow debugging with traps in notifiers
[*]   KGDB_KDB: include kdb frontend for kgdb
(0x1)   KDB: Select kdb command functions to be enabled by default
[*]     KGDB_KDB: keyboard as input device
(0)     KDB: continue after catastrophic errors
```

Figure 1

Configuration de Kgdb lors du menuconfig.

Il permet donc de jouer le rôle de serveur lors d'un débogage distant. Il offre pour cela deux modes de communication.

2.1 Port Serie

Afin de déboguer avec Kgdb, nous allons utiliser le driver **kgdboc**, pour *kgdb over console*. Ce dernier se configure très facilement via le **sysfs**. Nous allons définir le *tty* par lequel le driver recevra ses ordres du débogueur :

Terminal
```
$ echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc
```

ttyS0 représente le port série zéro pour le développement. Ceci peut s'avérer très pratique pour le développement embarqué.

Il suffit ensuite de connecter **gdb** via ce même port série :

Terminal
```
$ gdb ./vmlinux
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyS0
```

Il existe de multiples méthodes afin de mettre le noyau en pause afin qu'il attende une connexion de **gdb**. On peut soit passer la commande **kgdbwait** sur la ligne de commandes du noyau, soit une fois l'exécution lancée, utiliser un **sysrq-g**, soit un *Linux Magic System Request Key Hack* **[1]** :

Terminal

```
$ echo g > /proc/sysrq-trigger
```

En plus de stimuler le kgdb, ce système de *Magic Request* permet de déclencher des comportements tout à fait intéressants quand il s'agit de tester le noyau. Voici un petit extrait des commandes disponibles :

b	Reboote le système sans réaliser la synchronisation des systèmes de fichiers et le démontage des partitions
c	Permet de provoquer un crash du système par un dé-référencement mémoire
g	Stimule kgdb
i	Envoie un signal **SIGKILL** à tous les processus sauf **init**
e	Envoie un signal **SIGTERM** à tous les processus sauf **init**
p	Affiche l'ensemble des registres et *flags* sur la console
s	Tente une synchronisation de tous les systèmes de fichiers montés
u	Tente de remonter l'ensemble des systèmes de fichiers en lecture seule
t	Affiche une liste des tâches courantes
o	Éteint le système

Ce mécanisme est activable lors de la configuration du noyau (voir figure 2).

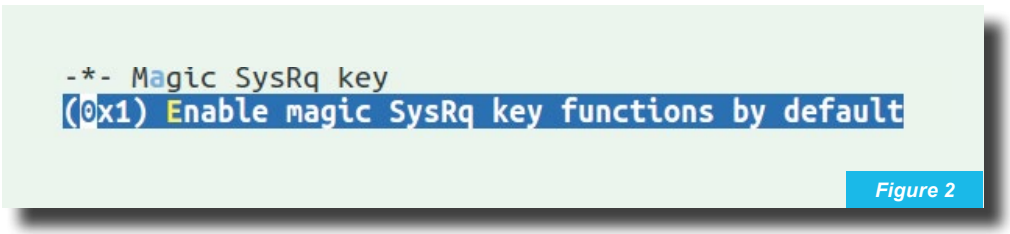


Figure 2

Configuration des SysReq dans menuconfig.

2.2 Ethernet

Il est possible de faire du débogage kernel, via kgdb, directement en Ethernet grâce au driver **kgdboe**, *kgdb over ethernet*. Pour ce faire, il faut que le driver Ethernet implémente l'API **NETPOLL**, et que le kernel soit compilé avec le support de NETPOLL. NETPOLL est une API kernel permettant à ce dernier d'offrir une pile

réseau simplifiée pour les modules noyau, tel que **kgdboe**, sans recours à la pile *userland* et bien sûr sans interruption. Au contraire de NETPOLL et **kgdboc**, **kgdboe** n'est pas disponible dans la *mainstream* du noyau, et doit donc être appliquée via le système de patch.

Afin de configurer **kgdboe**, on peut, comme toujours, soit le faire via la ligne de commandes du noyau grâce au paramètre **kgdboe**, soit dynamiquement via le système de fichiers **sys** :

Terminal
```
$ echo "@/,@10.0.2.2/" > /sys/module/kgdboe/paramters/kgdboe
```

Ce dernier suit la syntaxe suivante :

kgdboe=[src-port]@<src-ip>/[dev],[tgt-port]@<tgt-ip>/[tgt-macaddr]	
src-port	Source des paquets UDP (valeur par défaut 6443).
src-ip	IP source à utiliser.
dev	Interface réseau (valeur par défaut eth0).
tgt-port	Port du debogeur Gdb (valeur par defaut 6442).
tgt-ip	IP de la machine hébergeant Gdb.
tgt-macaddr	Adresse Ethernet hébergeant Gdb (valeur par défaut broadcasting).

Cette configuration se comprend aisément par la simplicité de la pile réseau interne au noyau. Dans l'exemple précédent, notre machine hébergeant notre Gdb doit donc se trouver impérativement sur le même sous-réseau que la machine exécutant le kernel.

3. QEMU

Il est possible de réaliser le débogage du noyau, sans module spécifique, mais en ajoutant une couche de virtualisation entre le débogueur et le processus noyau en cours d'exécution. Cette couche de virtualisation sera donc assurée par qemu, qui en plus de supporter le x86, supporte un grand nombre d'architectures, telles que ARM, MIPS ou encore SPARC.

Pour installer qemu pour x86 :

Terminal
```
$ sudo apt-get install qemu qemu-system
```

Pour virtualiser un noyau ARM :

Terminal
```
$ sudo apt-get install qemu qemu-system-arm
```

Il suffit ensuite d'installer la version de Gdb associée. Soit pour l'ARM, par exemple :

Terminal
```
$ sudo apt-get install gdb-arm-none-eabi
```

Cette technique, au contraire de la précédente, ne permet pas de lancer le noyau dans un mode d'attente du débogueur. Par contre, il est possible de demander à Qemu d'attendre la connexion d'un client Gdb avant de lancer l'exécution.

Il faut au préalable obtenir une image avec un *bootloader* simple, tel que **u-boot**, car le kernel ne fournit plus ce dernier dans ses sources.

Pour lancer l'image, il suffit donc de faire :

Terminal
```
$ qemu -gdb tcp::1234 -S <harddrive.img>
```

Il suffit ensuite de lancer gdb :

Terminal
```
gdb
(gdb) target remote localhost:1234
(gdb) add-symbol-file path/to/your/kernel
```

Il faut adapter la version de Gdb à la version de l'architecture ciblée.

CONCLUSION

Historiquement, le débogage noyau n'était pas chose aisée. Mais il était devenu nécessaire de fournir aux développeurs des moyens à la hauteur du défi que représente le développement de modules, drivers ou encore patches pour le noyau linux. Cela peut encore paraître, pour beaucoup de développeurs, être une tâche complexe, mais elle est grandement facilitée par des outils toujours plus performants. L'avènement de la virtualisation a aussi eu un impact positif sur le développement du noyau sur différentes architectures.

La première technique est obligatoire dès lors que vous devez composer avec un hardware exotique et donc que la virtualisation n'est pas possible. Elle sera donc privilégiée par la plupart des programmeurs de drivers. La seconde repose sur la souplesse de la virtualisation, et conviendra parfaitement aux développeurs désireux d'explorer les entrailles du noyau.

Je ne saurai trop vous conseiller de vous lancer dans le débogage d'un logiciel aussi passionnant que le noyau linux, ne serait-ce que par curiosité. ■

RÉFÉRENCE

[1] Linux Magic System Request Key Hack :
https://www.kernel.org/doc/Documentation/sysrq.txt

CAS N°4

MODIFICATION DES APPELS SYSTÈMES DU NOYAU LINUX :
MANIPULATION DE LA MÉMOIRE DU NOYAU

J.-M FRIEDT

Le noyau est une cible parfaite pour attaquer le système de par son accès à la mémoire et aux séquences d'instructions qui y sont lues pour être exécutées. Il est alors possible, à l'aide de méthodes de détournement des appels systèmes, de cacher à l'administrateur des traces d'une intrusion ou intercepter des opérations en cours d'exécution par les utilisateurs du système informatique attaqué.

Un logiciel est conçu pour effectuer certaines opérations, qui peuvent ou non convenir à nos intentions. Lorsque les sources du logiciel sont disponibles, rien n'est plus simple pour répondre à nos besoins : nous modifions les sources, recompilons le logiciel, et l'utilisateur est satisfait. Cependant, il existe des cas où les sources ne sont pas disponibles : logiciels propriétaires disponibles uniquement sous forme de binaire ou, une fois l'accès à un ordinateur sur lequel nous ne sommes pas administrateur acquis, noyau du système d'exploitation. Dans le cas particulier d'ordinateurs sous GNU/Linux, il se peut que nous ne désirions pas laisser de trace de notre passage, ou compliquer la tâche de l'administrateur cherchant à connaître la liste des utilisateurs et des fichiers sur son système. Le détournement des appels systèmes (`man syscalls`), en particulier tels que mis en œuvre dans les *rootkits*, répond à ces besoins. Plus intéressant que les mauvaises intentions de cacher un accès illégal à un ordinateur, manipuler le fichier binaire (exécutable **[1]** ou noyau) est l'opportunité de mieux comprendre le fonctionnement du système d'exploitation contrôlant notre environnement de travail, et d'en appréhender les forces et les faiblesses.

Presque tous les ouvrages contenant les mots « Linux » et « security » **[2][3]** discutent exclusivement des aspects d'administration – comment éviter une intrusion, depuis l'extérieur, d'un utilisateur non autorisé (*firewall* et protection réseau) ou comment éviter le gain de privilèges par un utilisateur autorisé qui n'est pas administrateur. Peu d'ouvrages discutent de la gestion du noyau et des appels systèmes qu'il fournit, puis de la façon de les manipuler **[4][5]**. Par ailleurs, la majorité des considérations sur ces sujets gravite autour des architectures compatibles Intel 32 ou 64 bits, mais la popularité d'Android **[6]** (basé sur un noyau Linux) et des plateformes conçues autour de processeurs ARM **[7]**, nous conduisent à appréhender les problèmes de détournement des appels système du noyau sur ces architectures. Tous les programmes exposés dans ce document ont été testés sur un ordinateur personnel basé sur un processeur Intel I5 exécutant **Debian/GNU Linux Sid** (noyau 4.6 x86_64) avec une compilation au moyen de **gcc** 6.1, ainsi que sur carte **Olinuxino-A13-micro** basée sur un processeur Allwinner A13 exécutant un système construit par buildroot **[8]** proposant Linux 4.4.2 et les outils associés dont gcc 4.9.3 en cross-compilateur pour ARM HF **[9]**. Les programmes proposés dans cette prose sont disponibles sur **http://jmfriedt.free.fr/lm_kernel.tar.gz**. Tous les messages issus du noyau sont affichés par `dmesg` : on notera que parfois l'*uptime* de l'ordinateur était très court. En effet, toute erreur dans un module noyau (par exemple accès à une zone mémoire non allouée) se traduira par une corruption du noyau et, tôt ou tard, un redémarrage du système. On évitera donc de tester ces programmes sur un ordinateur aux fonctionnalités critiques...

Les concepts modernes de sécurité informatique, en particulier tels qu'implémentés sur les systèmes compatibles Unix, tiennent en la séparation des droits. L'objectif est de restreindre au maximum les droits des utilisateurs, exposés au quotidien aux attaques virales **[10]**, *trojan* et autres *worm* **[11]**, et laisser un peu plus de droits à l'administrateur. En fin de compte, cette distribution des droits est toujours dévolue à un superviseur, qui dans le cas de Linux est pris en charge par le noyau monolithique qui donne son nom au système d'exploitation. Agir au niveau du noyau nous donne donc tous les pouvoirs sur le système informatique, mais surtout du point de vue du développeur sur systèmes embarqués et microcontrôleurs, nous redonne tous les droits d'accès aux ressources sans être bridés par un noyau qui nous surveille. En particulier, cette approche nous rappelle que tout processeur, du petit ARM7 aux gros multicœurs modernes qui équipent nos ordinateurs personnels, se résume en une unité arithmétique et logique qui reçoit ses instructions d'un gros tas d'octets qu'est la mémoire. Puisque la grande majorité des architectures actuellement en usage – à l'exception des architectures AVR d'Atmel et quelques petits Microchip conçus suivant l'architecture Harvard séparant mémoire d'instructions et mémoire de données – mélangent instructions et données selon les préceptes de Von Neuman, nous serons en droit de modifier (en manipulant des données) les instructions exécutées par le processeur.

Les développeurs au niveau du noyau connaissent bien cette problématique, et quelques tentatives de protéger les pages mémoire au moyen de l'unité de gestion de mémoire (MMU – *Memory Management Unit*) **[12]** essaient de nous brider, mais étant toujours maîtres absolus du système informatique au niveau du noyau, il nous suffira de désactiver ces protections pour laisser libre cours à nos activités. Dans les lignes qui vont suivre, nous verrons comment modifier la fonction appelée lors d'un appel système, ou modifier le contenu de cette fonction. Mais avant tout, qu'est-ce qu'un appel système ?

1. ASPECT HISTORIQUE : INT10, INT21 ET LEURS AMIS

Historiquement, les appels systèmes étaient gérés par des interruptions logicielles, une façon d'interrompre l'exécution séquentielle d'un programme pour sauter dans une fonction définie par le système. Sur les ordinateurs à base de processeurs Intel et compatibles, certains services sont fournis par le BIOS (en mode réel), accessibles par une série d'interruptions déclenchées par l'instruction `int`, avec par exemple l'interruption `0x10` pour les affichages sur écran, `0x13` pour les accès aux disques ou `0x16` pour accéder au clavier. Le choix du service est fourni dans le registre `AX`, et l'argument dans `BX`. Au-dessus du BIOS, DOS fournit ses propres appels système par l'interruption logicielle `0x21` (instruction assembleur `int 0x21`) après avoir chargé, toujours en mode réel, dans le registre `AX` le numéro du service et dans `BX` l'argument. On notera que bien avant l'avènement des BIOS propriétaires, l'ordinateur à base de 80286 commercialisé par IBM était fourni avec un manuel contenant non seulement les schémas électroniques de la carte mère et de ses périphériques, mais aussi les codes sources du BIOS et l'implémentation des services qu'il fournit ! (voir figure 1). Sous DOS, le concept d'interception des appels systèmes faisait partie de la programmation en *Terminate and Stay Resident* (TSR), dans laquelle le système d'exploitation devait ne pas libérer les ressources occupées par un programme, mais les conserver en mémoire pour y accéder sous interruption logicielle : **support.microsoft.com/en-us/kb/28568** propose un exemple sous MS-DOS d'interception des appels systèmes de gestion des répertoires. Cette approche se retrouve aujourd'hui, de façon considérablement enrichie puisque tous les appels systèmes qui définissent le respect de la norme POSIX **[13]** doivent être accessibles : dans le noyau Linux, la structure `syscall_table` fournit la liste des appels systèmes et la position (adresse) en mémoire de l'implémentation de chaque fonction. Cette table est ensuite utilisée pour générer les appels systèmes – donc en renseignant le registre `EAX` (mode protégé du processeur x86) et en appelant l'interruption `0x80`, tel que nous le constatons en consultant les sources du noyau `linux-4.4.2/arch/x86/ia32/ia32_signal.c`.

2. MODIFICATION DE L'ADRESSE DE LA FONCTION APPELÉE

Notre objectif est de modifier les appels systèmes. Nous pouvons dans un premier temps nous demander quel appel système est appelé au cours de quelle opération, pour évaluer l'intérêt de la tâche. `strace` (*trace system calls and signals*) est l'outil du shell qui informe l'utilisateur des appels systèmes sollicités lors de l'exécution d'un programme. Ainsi, lors de `strace ls`, nous constatons qu'une multitude d'appels à `open` sont nécessaires pour charger toutes les bibliothèques suite à l'exécution de `ls` par `execve`. Les affichages à l'écran à proprement parler se font par `write`, et finalement les ressources sollicitées sont relâchées par `close`.
Nous constatons donc que la manipulation de ces appels systèmes – `open`, `close`, `read` ou `write` – doit se faire avec le plus grand soin, au risque de complètement détruire les fonctionnalités du système d'exploitation. Avant de nous attaquer au problème de rediriger les appels systèmes, nous allons faire un petit détour par les pointeurs de fonction, pour nous rappeler comment rediriger un appel d'une fonction à une autre, et comment un appel de fonction est géré au niveau de l'assembleur.

Figure 1

```
 IBM Personal Computer MACRO Assembler   Version 2.00                1-16
ORGS ----- 04/21/86  COMPATIBILITY MODULE                            04-21-86

1547                                    PAGE
1548                                    ;-----  VECTOR TABLE
1549
1550                                    ;;-     ORG     0FEF3H
1551 1EF3                               ;       ORG     01EF3H          ;       AT LOCATION 0FEF3H
1552 1EF3                               VECTOR_TABLE    LABEL   WORD    ; VECTOR TABLE VALUES FOR POST TESTS
1553 1EF3 1EA5 R                                DW      OFFSET TIMER_INT        ; INT 08H - HARDWARE TIMER 0        IRQ  0
1554 1EF5 0987 R                                DW      OFFSET KB_INT           ; INT 09H - KEYBOARD                IRQ  1
1555 1EF7 0000 E                                DW      OFFSET  DTI             ; INT 0AH - SLAVE INTERRUPT INPUT
1556 1EF9 0000 E                                DW      OFFSET  DI1             ; INT 0BH -                         IRQ  3
1557 1EFB 0000 E                                DW      OFFSET  DI1             ; INT 0CH -                         IRQ  4
1558 1EFD 0000 E                                DW      OFFSET  DI1             ; INT 0DH -                         IRQ  5
1559 1EFF 0F57 R                                DW      OFFSET DISK_INT         ; INT 0EH - DISKETTE                IRQ  6
1560 1F01 0000 E                                DW      OFFSET  DI1             ; INT 0FH -                         IRQ  7
1561
1562
1563                                    ;-----  SOFTWARE INTERRUPTS  ( BIOS CALLS AND POINTERS )
1564 1F03 1065 R                                DW      OFFSET VIDEO_IO         ; INT 10H -- VIDEO DISPLAY
1565 1F05 184D R                                DW      OFFSET EQUIPMENT        ; INT 11H -- GET EQUIPMENT FLAG WORD
1566 1F07 1841 R                                DW      OFFSET MEMORY_SIZE_DET  ; INT 12H -- GET REAL MODE MEMORY SIZE
1567 1F09 0C59 R                                DW      OFFSET DISKETTE_IO      ; INT 13H -- DISKETTE
1568 1F0B 0739 R                                DW      OFFSET RS232_IO         ; INT 14H -- COMMUNICATION ADAPTER
1569 1F0D 1859 R                                DW      OFFSET CASSETTE_IO      ; INT 15H -- EXPANDED BIOS FUNCTION CALL
1570 1F0F 082E R                                DW      OFFSET KEYBOARD_IO      ; INT 16H -- KEYBOARD INPUT
1571 1F11 0FD2 R                                DW      OFFSET PRINTER_TO       ; INT 17H -- PRINTER OUTPUT
1572 1F13 0000                                  DW      00000H                  ; INT 18H -- 0F600H INSERTED FOR BASIC
1573 1F15 06F2 R                                DW      OFFSET BOOT_STRAP       ; INT 19H -- BOOT FROM SYSTEM MEDIA
1574 1F17 1E6E R                                DW      OFFSET TIME_OF_DAY      ; INT 1AH -- TIME OF DAY
1575 1F19 1F53 R                                DW      OFFSET DUMMY_RETURN     ; INT 1BH -- KEYBOARD BREAK ADDRESS
1576 1F1B 1F53 R                                DW      OFFSET DUMMY_RETURN     ; INT 1CH -- TIMER BREAK ADDRESS
1577 1F1D 10A4 R                                DW      OFFSET VIDEO_PARMS      ; INT 1DH -- VIDEO PARAMETERS
1578 1F1F 0FC7 R                                DW      OFFSET DISK_BASE        ; INT 1EH -- DISKETTE PARAMETERS
1579 1F21 0000                                  DW      00000H                  ; INT 1FH -- POINTER TO VIDEO EXTENSION
1580
1581 1F23                               SLAVE_VECTOR_TABLE      LABEL WORD      ; ( INTERRUPT 70H THRU 7FH )
1582
1583 1F23 0000 E                                DW      OFFSET RTC_INT          ; INT 70H - REAL TIME CLOCK         IRQ  8
1584 1F25 0000 E                                DW      OFFSET RE_DIRECT        ; INT 71H - REDIRECT TO INT 0AH     IRQ  9
1585 1F27 0000 E                                DW      OFFSET  DTI             ; INT 72H -                         IRQ 10
1586 1F29 0000 E                                DW      OFFSET  DI1             ; INT 73H -                         IRQ 11
1587 1F2B 0000 E                                DW      OFFSET  DI1             ; INT 74H -                         IRQ 12
1588 1F2D 0000 E                                DW      OFFSET INT_287          ; INT 75H -  -MATH COPROCESSOR      IRQ 13
1589 1F2F 0000 E                                DW      OFFSET  DIT             ; INT 76H -  -FIXED DISK            IRQ 14
1590 1F31 0000 E                                DW      OFFSET  DI1             ; INT 77H -                         IRQ 15
1591
1592                                    ;-----  DUMMY INTERRUPT HANDLER
1593
1594                                    ;;-     ORG     0FF53H
1595 1F53                               ;       ORG     01F53H
1596
1597 = 1F53                             DUMMY_RETURN    EQU     $       ; BIOS DUMMY (NULL) INTERRUPT RETURN
1598
1599 1F53 CF                                    IRET
1600
1601                                    ;-----  PRINT SCREEN
1602
1603                                    ;;-     ORG     0FF54H
1604 1F54                               ;       ORG     01F54H
1605 = 1F54                             PRINT_SCREEN    EQU     $
1606 1F54 E9 0000 E                             JMP     PRINT_SCREEN_1  ; VECTOR ON TO MOVED BIOS CODE
1607                                    .LIST                           ; TUTOR
1608                                    ;---------------------------------------
1609                                    ;                                      :
1610                                    ;       POWER ON RESET VECTOR          :
1611                                    ;                                      :
1612                                    ;---------------------------------------
1613                                    ;;-     ORG     0FFF0H
1614 1FF0                               ;       ORG     01FF0H
1615
1616                                    ;-----  POWER ON RESET
1617
1618 1FF0                               P_O_R   LABEL  FAR                      ; POWER ON RESTART EXECUTION LOCATION
1619
1620 1FF0 EA                                    DB      0EAH                    ; HARD CODE FAR JUMP TO SET
1621 1FF1 005B R                                 DW      OFFSET  RESET          ;  OFFSET
1622 1FF3 F000                                   DW      0F000H                 ;    SEGMENT
1623
1624 1FF5 30 34 2F 32 31 2F             DB      '04/21/86'              ; RELEASE MARKER
1625       38 36
1626
1627 1FFE                               ;       ORG     01FFEH
1628 1FFE FC                            DB      MODEL_BYTE              ; THIS PC'S ID ( MODEL BYTE )
1629
1630 1FFF                               CODE    ENDS                    ; CHECKSUM AT LAST LOCATION
1631                                    END
```

Scan de la documentation du PC AT d'IBM à base de 80286, incluant le listing du BIOS et notamment la liste des interruptions, incluant les interruptions logicielles faisant office d'appels système. Une copie du document complet est disponible sur http://bitsavers.trailing-edge.com/pdf/ibm/pc/at/1502494_PC_AT_Technical_Reference_Mar84.pdf.

ARCHITECTURES HARVARD ET VON NEUMANN

L'hypothèse que nous ferons tout au cours de cette étude est qu'une instruction chargée de manipuler une donnée est capable de modifier une instruction qui sera exécutée par le processeur. Cette hypothèse n'est pas évidente, et valable uniquement sur une architecture de type Von Neumann. À la genèse du développement des ordinateurs alors que les processeurs, tels que l'IBM ASCC installé à Harvard, étaient tellement lents qu'il fallait plusieurs secondes pour effectuer une opération arithmétique, un gain de temps consistait à chercher les instructions dans une mémoire, et les données dans une autre mémoire séparée : les deux bus d'accès à ces ressources sont séparés et accessibles simultanément, au lieu d'un accès séquentiel comme dans l'architecture Von Neumann où données et instructions occupent la même mémoire. Aujourd'hui, les seuls processeurs à encore être conçus en architecture Harvard sont les AVR de Atmel et les PIC de la gamme Microchip. Ce point n'est pas anodin et a des conséquences sur la capacité d'un compilateur à optimiser l'utilisation des ressources. Un cas classique concerne les polices de caractères : tableaux volumineux de données qui ne sont accédées qu'en lecture, les tableaux définissant les polices de caractères sont typiquement préfixés de l'attribut `const` pour indiquer à gcc de placer les données en mémoire non-volatile, généralement disponibles en excès, contrairement à la mémoire volatile. Ainsi, `const police[95*8]=0;` pour définir les 95 caractères ASCII affichables se traduit sur (architecture Von Neuman) MSP430 par (`msp430-size` après avoir compilé par `msp430-gcc`) une occupation de 1630 octets de flash (section `text`) et 2 octets de RAM (sections `bss` et `data`). Au contraire sur AVR, `avr-size` nous indique que la même compilation se traduit par 38 octets de `text` et 1520 octets de `data` : un tel programme tient tout juste dans un Atmega16U4 alors que ses 16 KB de flash restent inoccupés. Sur ordinateur personnel, ces problèmes de séparation de données et instructions ne devraient pas survenir, bien que la gestion des caches entre la mémoire généraliste et les processeurs puisse devenir un problème : en cas de dysfonctionnement des exemples proposés, on pourra tenter d'invalider les caches pour forcer le processeur à recharger les instructions en mémoire [14].

2.1 En espace utilisateur

Pour appréhender l'approche consistant à modifier l'adresse de la fonction appelée **[15]**, il est bon de se remémorer le concept de pointeur de fonction. Appeler une fonction revient à empiler les arguments, sauter (instruction assembleur `call`) à l'adresse de la fonction chargée du traitement, puis revenir au programme principal en dépilant la valeur du *program counter* qui avait été empilée au moment de l'appel de la *subroutine*. C'est à cette dernière étape que nous pouvons éventuellement induire un saut dans une fonction tierce en ayant corrompu la pile dans la fonction appelée dans l'attaque classique du *buffer overflow*. Ici nous nous intéressons à modifier l'adresse de la fonction appelée. Dans l'exemple ci-dessous, *ma_func* est un pointeur vers une fonction qui n'est pas déclarée initialement :

Fichier

```
#include <stdio.h>
int toto(int x) {return (x+1);}
int tata(int x) {return (x+2);}

int main()
{int (*ma_func)(int);
 ma_func = &toto; printf("%d\n",ma_func(1));
 ma_func = &tata; printf("%d\n",ma_func(1));
 return 0;
}
```

Ayant déclaré le pointeur de fonction, mais sans l'avoir initialisé, appeler *ma_func(argument)* se traduit évidemment par une erreur de segment (*segmentation fault*) puisque nous induisons un saut vers un segment mémoire qui n'est pas attribué au programme en cours d'exécution. Nous pouvons alors assigner le pointeur `ma_func(argument)` à l'adresse d'une fonction du programme, ici `toto` puis `tata`, qui se traduira bien par l'effet recherché, ici un résultat de `2` puis de `3`. Nous serons donc capables de rediriger un appel en modifiant l'argument de l'instruction `call` lors du saut au sous-programme. Cette méthode est parfois utilisée lorsqu'une même fonction se décline de différentes façons selon les périphériques auxquels elle s'applique : nous avons par exemple rencontré ce cas dans l'implémentation libre du bus **Modbus** pour microcontrôleur disponible sur **http://www.freemodbus.org/**. Dans ce programme, une méthode commune à toutes les implémentations d'un protocole, par exemple `write()` ou `open()`, est définie, et les diverses implémentations sont déclinées pour les diverses implémentations matérielles en faisant pointer chaque fonction vers l'implémentation appropriée. Dans freemodbus, la fonction `eMBInit()` de `mb.c` initialise les pointeurs de fonctions selon le protocole de communication utilisé. Si au lieu d'assigner statiquement à la compilation la fonction appelée nous désirons l'assigner dynamiquement au cours de l'exécution, l'adressage indirect permet d'appeler une adresse fournie dans un registre. Dans l'assembleur AVR, plus facile à lire que l'assembleur x86, cette différence s'observe entre l'appel direct et par pointeur de fonction à `toto()` :

Fichier

```
#include <stdio.h>
int toto(int x) {return (x+1);}

int main()
{int (*ma_func)(int);
 printf("%d\n",toto(1));                    // appel direct à toto
 ma_func = &toto; printf("%d\n",ma_func(1)); // passage par pointeur de
fonction
 return 0;
}
```

que nous compilons avec l'option de déverminage `-g` pour conserver les symboles :

Terminal

```
int toto(int x) {return (x+1);}
  ea:   cf 93           push    r28
[...]
  fc:   01 96           adiw    r24, 0x01       ; 1
[...]
 104:   cf 91           pop     r28
 106:   08 95           ret

int main()
{[...]
 printf("%d\n",toto(1));
 130:   81 e0           ldi     r24, 0x01       ; 1
 132:   90 e0           ldi     r25, 0x00       ; 0
 134:   0e 94 75 00     call    0xea    ; 0xea <toto>
[...]
ma_func = &toto; printf("%d\n",ma_func(1));
[...]
 168:   f9 01           movw    r30, r18
 16a:   09 95           icall
[...]
}
```

Le code assembleur est issu de `avr-objdump -dSt a.out`. Dans ce cas, le registre `Z` contient l'adresse de la fonction appelée, tel que décrit dans la liste des mnémoniques de l'assembleur AVR [16]. `Z` est la concaténation de `R31` et `R30`, d'où notre intérêt pour la manipulation de `R30` juste avant l'appel à `icall`.

2.2 En espace noyau

Ayant compris que l'adresse mémoire contenant la fonction peut être modifiée, il devient presque trivial d'attribuer une nouvelle destination à un appel système. Nous avons vu que la liste des appels systèmes sollicités par une commande shell est affichée par `strace`. Plutôt que risquer de casser notre système en manipulant un appel critique, nous allons considérer une fonction qui ne sert presque à rien, mais dont l'effet est spectaculaire, la création de répertoires. Lorsque nous lançons `strace mkdir toto`, nous obtenons notamment :

Terminal
```
mkdir("toto", 0777)                     = 0
```

Nous constatons que l'appel système `mkdir` est utilisé pour créer le répertoire : c'est lui que nous allons détourner.

L'approche consistant à modifier le pointeur vers la fonction implémentant un appel système est tellement évidente que les développeurs du noyau Linux essaient de brider un peu notre capacité à modifier ces appels systèmes en n'exportant pas le symbole de la table contenant les adresses des appels systèmes [17] : ne connaissant pas l'adresse où se trouve la liste des appels systèmes, il devrait nous être impossible de la modifier. Il faut donc d'abord identifier l'emplacement de la table des symboles en mémoire [18], avant d'avoir le droit de la modifier [19].

Afin de trouver l'emplacement de la table des appels systèmes, il devrait suffire de balayer la mémoire à la recherche de l'adresse de l'appel système que nous cherchons à modifier. La méthode n'est pas garantie, car il se pourrait qu'un emplacement mémoire contienne par hasard les valeurs correspondant à l'adresse d'un appel système, mais les chances sont faibles et cette approche fonctionne dans la pratique. Cependant, tous les appels systèmes ne sont pas exportés vers le noyau : nous devons en trouver un afin de connaître l'adresse à rechercher en mémoire. La liste des appels système se trouve dans `include/linux/syscalls.h` des sources du noyau : tous les appels systèmes commencent par `sys_`. Un développeur naïf pourrait se dire que voulant manipuler `mkdir`, il nous suffit de trouver l'adresse de l'appel système `sys_mkdir` et de trouver son emplacement en mémoire. Cette tentative échoue au moment de l'édition de lien lors de la compilation du noyau, avec un message nous informant que le symbole `sys_mkdir` n'est pas connu. En effet, l'appel système est implémenté, mais le symbole n'est pas exporté vers les autres modules du noyau. Nous devons donc compléter la recherche pour identifier la liste des symboles exportés qui correspondent à nos besoins : dans les sources du noyau (ici 4.4.2) nous avons :

Terminal
```
linux-4.4.2$ grep -r EXPORT_SYMBOL * | grep \(sys_
[...]
fs/open.c:EXPORT_SYMBOL(sys_close);
kernel/time/time.c:EXPORT_SYMBOL(sys_tz);
```

Ceci nous indique que seul **sys_close** est exporté. Nous allons donc rechercher l'occurrence de ce symbole, en sachant qu'ensuite la table est arrangée dans le même ordre que la liste des appels définie dans **/usr/include/x86_64-linux-gnu/asm/unistd_64.h** :

Fichier

```
#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
[...]
#define __NR_mkdir 83
```

Il y a donc 80 emplacements mémoire d'écart entre l'adresse de l'appel système **close** et **mkdir** dans la table des appels systèmes. Le module suivant effectue la recherche dans la mémoire allouée au noyau, qui commence à l'adresse définie par la constante **PAGE_OFFSET** tel que décrit dans **linux-4.4.2/Documentation/x86/x86_64/mm.txt** : il s'agit de la constante ajoutée à l'adresse de la mémoire physique pour définir l'adresse virtuelle de la mémoire dédiée au noyau (**https://linux-mm.org/VirtualMemory**).

Fichier

```
#include <linux/module.h>
#include <linux/syscalls.h>

static unsigned long* cherche_table(void)
{unsigned long int offset = PAGE_OFFSET;   // début du kernel en RAM
 unsigned long *sct;
 printk(KERN_INFO "PAGE_OFFSET=%lx\n",PAGE_OFFSET);
 while (offset < ULLONG_MAX)     // recherche ds la mémoire allouée au
noyau
  {sct = (unsigned long *)offset;
   if (*sct == (unsigned long)&sys_close)  // cherche l'@ de sys_close
      return sct;                          // on a trouvé l'@ de début
   offset += sizeof(void *);
  }
 return NULL;
}

static int __init module_start(void)
{unsigned long *addr_table;
 addr_table = cherche_table();
 printk(KERN_INFO "table: %lx\n",(unsigned long)addr_table);
 printk(KERN_INFO "NR close: %d\n",__NR_close);
 printk(KERN_INFO "NR mkdir: %d => %lx",__NR_mkdir,addr_table[__NR_
mkdir-__NR_close]);
 printk(KERN_INFO "*void: %ld\n",sizeof(void*));
 return 0;
}

static void __exit module_end(void) {}

module_init(module_start);
module_exit(module_end);
MODULE_LICENSE("GPL");
```

La fonction `cherche_table()` balaie la mémoire du noyau à partir de `PAGE_OFFSET` jusqu'à trouver un emplacement contenant l'adresse de l'appel système `sys_close` : nous avons vu auparavant avec les pointeurs de fonctions que l'adresse de la fonction s'obtient en préfixant son nom par `&`. Si le contenu de l'emplacement pointé par `sct` contient l'adresse `&sys_close`, il est probable que nous ayons trouvé la table des appels systèmes. Nous renvoyons donc cette adresse à la fonction initialisant le pilote `module_start()` et y affichons l'emplacement de la table qui doit correspondre à l'emplacement de `sys_call_table`, le contenu de l'emplacement mémoire que nous avons trouvé et le contenu de l'emplacement 80 cases mémoire plus loin, puisque `__NR_mkdir-__NR_close` vaut `80`. Chaque emplacement mémoire contient une adresse, donc de taille `sizeof(void*)`, ou 8 octets sur une architecture 64 bits.

Le chargement de ce module dans un noyau 4.6 (Debian Sid à la date de rédaction de cette prose) sur une architecture 64 bits compatible Intel (x86_64) se traduit par les messages suivants :

Terminal
```
[100266.370251] PAGE_OFFSET=ffff880000000000
[100266.433568] table: ffff8800016001f8
[100266.433570] NR close: 3
[100266.433571] NR mkdir: 83 => ffffffff812020d0
[100266.433572] *void: 8
```

Ceux-ci indiquent l'adresse de début de la zone mémoire allouée au noyau (constante `PAGE_OFFSET`), l'emplacement de la table des symboles identifiée par la recherche de l'adresse de l'appel système `sys_close()`, et le contenu de l'emplacement mémoire qui se trouve 80 cases après l'adresse que nous avons ainsi identifiée. Nous validons la cohérence de ces informations avec celles fournies par le noyau dans `/boot/System.map` ou dans `/proc/kallsyms`. En effet :

Fichier
```
/boot/System.map-4.6.0-1-amd64: ffffffff816001e0 R sys_call_table
/proc/kallsyms                : ffffffff816001e0 R sys_call_table
```

Ceci indique que la table des appels système se trouve à l'adresse `0x816001e0`, soit 24 octets avant l'adresse que nous avons identifiée, ce qui est cohérent avec l'indice de l'appel système (3) multiplié par la taille de chaque case mémoire (8 octets). Par ailleurs, nous vérifions que l'emplacement mémoire que nous avons identifié comme contenant l'appel système `mkdir` contient la bonne information : `/proc/kallsyms` nous informe que :

Fichier
```
ffffffff812020d0 T sys_mkdir
```

Ce qui est bien l'adresse trouvée par le pilote noyau.

Nous voilà donc convaincus de la capacité à identifier l'emplacement de la table des appels systèmes en mémoire et l'adresse de chaque fonction appelant ces appels système. La dernière subtilité pour arriver à nos fins tient encore à la MMU, qui interdit d'écrire dans la page allouée aux appels systèmes (tel qu'indiqué par l'attribut `R` dans `/proc/kallsyms`). Nous débloquons cet accès en informant la MMU de désactiver la protection en manipulant le bit `WP` du registre `CR0` [20] (fonctions à évidemment retirer lors des tests sur architecture ARM) :

Fichier

```
#include <linux/module.h>
#include <linux/syscalls.h>

unsigned long *addr_table;   // global pour passer à exit()
#ifndef __ARMEL__
unsigned long original_cr0;  // global pour passer à exit()
#endif

// cf http://www.csee.umbc.edu/courses/undergraduate/CMSC421/fall02/burt/
projects/howto_add_systemcall.html
asmlinkage        // passage de paramètres par la pile et non par les
registres
long (*ref_sys_mkdir)(const char __user*,int); // proto (dépend de la
fonction)

asmlinkage
long mon_mkdir(const char __user *pnam, int mode)
{printk(KERN_INFO "intercept: %s:%x\n", pnam,mode); return 0;}

static unsigned long* cherche_table(void)
{[...]}                      // recherche de la table des appels sys

static int __init module_start(void)
{
 addr_table = cherche_table();
 [...]                       // affichage des informations
#ifndef __ARMEL__
 original_cr0 = read_cr0();  // passe la page des appels sys en écriture
 write_cr0(original_cr0 & ~0x00010000); // bit 16 est WP: on met à 0
#endif
 ref_sys_mkdir = (void *)addr_table[__NR_mkdir-__NR_close];
 addr_table[__NR_mkdir-__NR_close] = (unsigned long)mon_mkdir;
#ifndef __ARMEL__
 write_cr0(original_cr0);    // repasse la page des appels en lecture
seule
#endif
 return 0;
}

static void __exit module_end(void)
{
#ifndef __ARMEL__
 write_cr0(original_cr0 & ~0x00010000);  // remet la fonction originale
#endif
 addr_table[__NR_mkdir-__NR_close] = (unsigned long)ref_sys_mkdir;
#ifndef __ARMEL__
 write_cr0(original_cr0);
#endif
 printk(KERN_INFO "bye\n");
}
```

Les messages au chargement du pilote restent les mêmes, mais cette fois lorsque nous tentons de créer un répertoire par **`mkdir toto`** après chargement du module, nous recevons le message additionnel :

Terminal

```
[103669.045129] intercept: toto:1ff
```

Celui-ci indique que nous avons bien intercepté l'appel système. Évidemment, en l'état, un utilisateur du système se rendra immédiatement compte de l'interception de l'appel système puisque le répertoire n'est pas créé, et nous pouvons cacher les traces de l'interception en appelant tout de même la fonction d'origine pour effectuer le travail :

Fichier

```
asmlinkage
long new_sys_mkdir(const char __user *pnam, int mode)
{long ret=ref_sys_mkdir(pnam, mode); // cette fois on crée le répertoire
 printk(KERN_INFO "intercept %ld: %s:%x\n", getuid(),pnam,mode);
 return 0;
}
```

Nous avons donc démontré la capacité à intercepter les appels systèmes, donner l'impression que l'appel a abouti tout en manipulant éventuellement le résultat. Cette approche est classiquement utilisée dans les *rootkits* pour cacher les traces de l'accès : le meilleur moyen de cacher ses traces est de faire croire au système d'exploitation que les fichiers n'existent pas, en interceptant **`getdents`** qui est la fonction utilisée par **`ls`** pour afficher le contenu d'un répertoire, et retirer les entrées des fichiers ou répertoires que nous ne voulons pas afficher :

Terminal

```
# strace ls /tmp/ |& grep -A2 tmp
[...]
stat("/tmp/", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=740, ...}) = 0
open("/tmp/", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFDIR|S_ISVTX|0777, st_size=740, ...}) = 0
getdents(3, /* 37 entries */, 32768)    = 1368
[...]
```

3. MODIFICATION DU CONTENU DE L'EMPLACEMENT MÉMOIRE

Nous avons vu comment modifier la fonction appelée lors d'un appel système. Une alternative à cette approche consiste à modifier le contenu de la fonction appelée, par exemple en écrasant le contenu de la mémoire pointé par la table des appels systèmes. Cette approche est celle classiquement implémentée par les virus [21][22][23] , qui écrasent le début du fichier infecté. Dans le meilleur des cas, le bout de code écrasé est déplacé en fin d'exécutable pour y sauter une fois l'infection achevée et donner l'impression à l'utilisateur que son programme est toujours fonctionnel. Dans le cas du noyau Linux, nous verrons que l'excellente optimisation de gcc rend cette approche peu intuitive, car des fonctions que nous pourrions croire autonomes d'après le code source sont insérées directement dans le flux d'exécution (*inline*) voire disparaissent complètement, car leur résultat est connu du compilateur. Nous allons néanmoins tenter de trouver quelques exemples pour illustrer cette approche.

3.1 En espace utilisateur

Rappelons une fois de plus que pour une mémoire d'ordinateur, le concept de donnée ou d'instruction n'existe pas : la RAM est un tas d'octets, que l'on peut soit exécuter si la valeur correspond à un opcode définissant une opération de l'unité arithmétique et

logique, soit traiter comme un argument d'une opération arithmétique ou logique. Ainsi, rien n'interdit d'écrire une valeur dans un emplacement mémoire, pour ensuite l'exécuter (concept que les défenseurs des langages protégeant la mémoire tels que **Java** interdisent, mais qui au contraire fera ici toute la beauté du **C** et de la manipulation de la mémoire au travers des pointeurs).

Dans l'exemple ci-dessous, nous abordons deux cas : écraser une fonction avec le contenu d'une autre fonction en mémoire, ou placer une fonction dans la pile puis y sauter pour en exécuter le contenu. Ces approches étant des sources classiques d'attaques, Linux tente désormais de s'en protéger en interdisant l'écriture dans les pages mémoire contenant du code – bridant par la même occasion le code auto-modifiable – ou l'exécution des instructions placées sur la pile. Nous devons donc débloquer ces fonctions au moyen de **`mprotect()`** afin d'autoriser l'écriture et l'exécution sur ces plages de mémoire. Les deux options du programme, **`function_overwrite`** ou **`stack_overwrite`**, sont exclusives et activent l'une des deux approches proposées ci-dessus :

Fichier

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h> // mprotect
#include <unistd.h>   // sysconf

#define function_overwrite  // écrase fonction() avec fonction_vide()
#undef  stack_overwrite     // écrase un tableau de la pile avec fonction_
vide appelée par ma_func

int i=0;
void fonction(){printf("fonction\n");} // la fonction originale
void fonction_vide() {i=i+1;}          // la fonction de remplacement

int main()
{char c[128];
 int pagesize;
 void (*ma_func)(void);
 int longueur=(int)&main-(int)&fonction_vide;
 printf("%d %x %x %d %d\n",i,&fonction,&fonction_vide,(int)&fonction_vide-
(int)&fonction,longueur);
 fonction();
 pagesize = sysconf(_SC_PAGE_SIZE);     // mprotect doit être aligné
#ifdef function_overwrite
 mprotect((void*)((int)&fonction&~(pagesize-1)),(int)&main-
(int)&fonction,PROT_EXEC|PROT_READ|PROT_WRITE);
 memcpy(&fonction,&fonction_vide,longueur); // écrase l'ancienne fonction
#else
 // l'exécution d'un code sur la pile est aussi possible par gcc -z
execstack
 mprotect((void*)((int)c&~(pagesize-1)),longueur,PROT_EXEC|PROT_READ|PROT_
WRITE);
 memcpy(c,&fonction_vide,longueur);         // place la nouvelle fonction
sur la pile
 ma_func=(void *)c; ma_func();
#endif

 fonction();
 printf("i=%d\n",i);
}
```

Nous constatons qu'effectivement l'appel à `fonction()` en début de programme se traduit par l'affichage du message « fonction », alors qu'après la manipulation, ce même appel, en fin de programme, se traduit par l'absence de message, mais la modification de la variable `i`. Par ailleurs, l'absence de distinction entre donnée et instruction est illustrée par la fonction `ma_func()` qui pointe vers le tableau d'octets `c` dans lequel nous avons copié la séquence d'opcodes `fonction_vide()`.

3.2 En espace noyau

Ayant introduit le concept en espace utilisateur, pouvons-nous le transposer au noyau ? Ici encore, le gestionnaire de mémoire va nous poser soucis, en interdisant l'accès aux pages mémoires inutilisées. Il semble évident que manipuler le contenu de la mémoire rend le système excessivement vulnérable, et depuis longtemps des protections ont été mises en place pour éviter l'accès à l'ensemble de la mémoire par un utilisateur **[24]**, même possédant les droits d'administration : **http://lwn.net/Articles/267427/** décrit les limitations mises en place sur `/dev/mem` dès 2008. Un pilote noyau n'est évidemment pas assujetti à ces restrictions, et pourra librement sonder la mémoire tant que la MMU l'y autorise.

3.3 Écraser une fonction par une autre

Dans l'exemple ci-dessous, la `fonction1` va être écrasée par `fonction2`. Pour rendre l'exemple simple à comprendre, ces deux fonctions sont excessivement simples – tellement simples que gcc veut absolument les optimiser en pré-calculant le résultat à la compilation. Nous sommes obligés de forcer gcc à oublier ses ambitions d'optimisation en lui interdisant de déplacer les fonctions dans le code appelant (`noinline`) et en lui interdisant de faire d'hypothèse sur une valeur connue de la variable fournie en argument (`volatile`) :

Fichier

```
#include <linux/module.h>

static noinline int fonction1(volatile int i)
   {// printk(KERN_INFO "fonction1");
    return(i+1);}
static noinline int fonction2(volatile int i) { return(i+2);}
static noinline int fonction3(volatile int i) { return(i+3);}

void difference(char *c1,char *c2,int l) // vérifie la diff entre deux
zones mémoire
{int k,f=0;
 for (k=0;k<l;k++)
     if (c1[k]!=c2[k])
        {f++;
         printk(KERN_INFO "diff %lx: %hhx %hhx",
                (unsigned long)(c1+k),(char)c1[k],(char)c2[k]);
        }
 if (f==0) printk(KERN_INFO "no difference");
}

void overwrite(void)
{char *sct,l;
```

```
 char *c1=(char*)&fonction1,*c2=(char*)&fonction2,*c3=(char*)&fonction3;
#ifndef __ARMEL__
 unsigned long original_cr0;
 original_cr0 = read_cr0();              // autorise écriture
 write_cr0(original_cr0 & ~0x00010000);
#endif
 sct = (char*) fonction1;
 printk(KERN_INFO "sct=%lx\n",(unsigned long)sct);
 printk(KERN_INFO "fonction2=%lx\n",(long)c2);
 printk(KERN_INFO "fonction3=%lx\n",(long)c3);
l=(unsigned long)c3-(unsigned long)c2; // longueur de fonction2

 printk(KERN_INFO "longueur=%x\n",(unsigned int)l);
 difference((char*)c1,(char*)c2,l);
 printk(KERN_INFO "res avant fonction1(1)=%d",fonction1(1));

 // ecrase fonction1 avec fonction2
 memcpy((void*)c1,(void*)c2,((unsigned long)c3-(unsigned long)c2));

 difference((char*)c1,(char*)c2,l);
 printk(KERN_INFO "res apres fonction1(1)=%d",fonction1(1));
 printk(KERN_INFO "the end\n");
#ifndef __ARMEL__
 write_cr0(original_cr0);          // interdit écriture
#endif
}

static int __init module_start(void) {overwrite();return(0);}

static void __exit module_end(void) {}

module_init(module_start);
module_exit(module_end);
MODULE_LICENSE("GPL");
```

Tout ceci se traduit par :

Terminal

```
[117421.948195] sct=ffffffffc0a71000
[117421.948198] fonction2=ffffffffc0a71020
[117421.948199] fonction3=ffffffffc0a71040
[117421.948199] longueur=20
[117421.948200] diff ffffffffc0a71017: 1 2
[117421.948201] res avant fonction1(1)=2
[117421.948202] no difference
[117421.948203] res apres fonction1(1)=3
[117421.948203] the end
```

Une alternative pour éviter les optimisations abusives de gcc, observables par **objdump -dSt** du fichier **.ko** généré, est de faire appel à des fonctions plus complexes, faisant par exemple intervenir **printk**. Nous constatons bien que le premier appel à **fonction1(1)** renvoie **2**, comme prévu. Nous observons la différence entre **fonction1** et **fonction2** qui est une différence sur l'argument de la somme, de **1** à **2**, puis l'absence de différence après avoir écrasé **fonction1**. Finalement, le second appel à **fonction1** renvoie **3**, comme c'eut été le cas si nous avions appelé **fonction2**. L'écrasement de **fonction1** par **fonction2** est démontré.

3.4 Identifier l'emplacement d'une fonction

Nous sommes certes capables d'écraser une fonction dans le noyau, mais le problème tient maintenant à savoir où se trouve la fonction que nous désirons écraser. Comme auparavant, nous pouvons tenter une recherche sur l'ensemble de la mémoire. Dans l'exemple ci-dessous, nous utilisons une signature unique au module testé – la présence de la chaîne de caractères **qwertyuiop** qui a peu de chances de se trouver en mémoire par ailleurs – pour trouver l'emplacement de notre pilote :

Fichier

```
#include <linux/module.h>
long cherche(unsigned long offset)
{unsigned char sct,sctp1,sctp2,sctp3,sctp4,sctp5;
 char s[15]="qwertyuiop\0";
 printk(KERN_INFO "%s\n",s);

 while (offset < ULLONG_MAX)     // recherche dans la mémoire allouée au
noyau
   {sct  = *(unsigned char*)offset;
    sctp1= *(unsigned char*)(offset+1);
    sctp2= *(unsigned char*)(offset+2);
    sctp3= *(unsigned char*)(offset+3);
    sctp4= *(unsigned char*)(offset+4);
    sctp5= *(unsigned char*)(offset+5);
    if ((sct=='q') && (sctp1=='w') && (sctp2=='e') && (sctp3=='r'))
    // if ((sct=='j') && (sctp1=='m') && (sctp2=='f'))
       {printk(KERN_INFO "-> %lx: %x %x %x %x %x %x\n",(unsigned long)offs
et,sct,sctp1,sctp2,sctp3,sctp4,sctp5);
        return(offset);
       }
    offset++;
   }
 return offset;
}

static int __init module_start(void)
{printk(KERN_INFO "%lx\n",PAGE_OFFSET);
 printk(KERN_INFO "cherche: %lx\n",(unsigned long)&cherche);
 cherche(PAGE_OFFSET);
 cherche((unsigned long)(&cherche));
return(0);}

static void __exit module_end(void) {}

module_init(module_start);
module_exit(module_end);
MODULE_LICENSE("GPL");
```

Terminal

```
[118418.835824] ffff880000000000
[118418.835827] cherche: ffffffffc0a71000
[118418.835828] qwertyuiop
[118419.318922] -> ffff8800000f4af1: 71 77 65 72 74 79
[118419.318924] qwertyuiop
[118419.318929] -> ffffffffc0a7205e: 71 77 65 72 74 79
```

Nous constatons donc que la signature que nous avons inséré dans notre pilote se retrouve en plusieurs endroits dans la mémoire allouée au pilote, soit en début de mémoire si la recherche commence à **PAGE_OFFSET**, soit bien plus loin si nous commençons la recherche à l'adresse de la première fonction du module. La première occurrence correspond probablement à la zone allouée pour stocker les variables du noyau et non au module lui-même.

3.5 Modifier le résultat d'une fonction

Ici nous avons introduit manuellement la signature, mais serions-nous capables de modifier le résultat d'une opération en modifiant la séquence d'opcodes ? Le cas est très similaire au précédent puisque données et opcodes sont manipulés indifféremment par le processeur :

Fichier

```
#include <linux/module.h>

noinline int adieu(int);

long cherche(void)
{unsigned char sct,sctp1,sctp2,sctp3,sctp4;
 unsigned long int offset = (unsigned long)&cherche; // 0xffffffffc0000000
début du kernel en RAM
 printk(KERN_INFO "%lx\n",PAGE_OFFSET);
 while (offset < ULLONG_MAX)   // recherche ds la mémoire allouée au noyau
   {if ((offset%0x1000000)==0x0)
        printk(KERN_INFO "ok : %lx\n",(unsigned long)offset);
    sct  = *(unsigned char*)offset;
    sctp1= *(unsigned char*)(offset+1);
    sctp2= *(unsigned char*)(offset+2);
    sctp3= *(unsigned char*)(offset+3);
    sctp4= *(unsigned char*)(offset+4);
#ifndef __ARMEL__
 if ((sct==0x8d) && (sctp1==0x87) && (sctp2==0x9a) && (sctp3==0x02) &&
(sctp4==0x00))
#else
 if ((sct==0xe2) && (sctp1==0x80) && (sctp2==0x0f) && (sctp3==0xa6) &&
(sctp4==0xe2))
#endif
    {printk(KERN_INFO "fonction cherche @ %lx\n",(unsigned long)&cherche);
     printk(KERN_INFO "code offset %lx\n",(unsigned long)offset);
     return(offset);
    }
    offset++;
   }
 return offset;
}

static int __init module_start(void)
{unsigned long offset;
#ifndef __ARMEL__
 unsigned long original_cr0;
 char s[4]={0x8d,0x87,0x2b,0x02};  // nouvelle séquence d'opcodes x86
#else
 char s[4]={0x8a,0x0f,0x80,0xe2};  // nouvelle séquence d'opcodes ARM
// initialement e2800fa6 pour add r0, r0, #664 ; 0x298
#endif
```

```
 offset=cherche();
#ifndef __ARMEL__
original_cr0 = read_cr0();  // passe la page des appels sys en écriture
 write_cr0(original_cr0 & ~0x00010000);
#endif
 memcpy((char*)offset,s,4);
#ifndef __ARMEL__
 write_cr0(original_cr0);  // passe la page des appels en lecture seule
#endif
 return(0);
}

noinline int adieu(int k) {return(k+666);}

static void __exit module_end(void)
{printk(KERN_INFO "sortie du module : %d\n",adieu(0));}  // doit renvoyer 0+666

module_init(module_start);
module_exit(module_end);
MODULE_LICENSE("GPL");
```

Si nous commentons les fonctions de modification du contenu de la fonction **`adieu()`**, quitter le pilote se traduit bien par l'affichage de 666, qui est le résultat attendu de l'appel à **`adieu(0);`**. Comment avons-nous effectué la recherche de la séquence d'opcodes et les modifications à faire ? **`objdump -dSt module.ko`** (ou pour la version ARM, **`arm-buildroot-linux-uclibcgnueabihf-objdump`**) fournit la séquence de mnémoniques et d'opcodes associés à un exécutable désassemblé. Pour Intel, nous trouvons :

Terminal

```
a0:   e8 00 00 00 00          callq  a5 <adieu+0x5>
a5:   8d 87 9a 02 00 00       lea    0x29a(%rdi),%eax
ab:   c3                      retq
```

Où nous retrouvons bien **`0x29a=666`** (que nous remplaçons par **`0x22b=555`**), tandis que pour ARM nous observons :

Terminal

```
cc:   e2800fa6        add     r0, r0, #664    ; 0x298
d0:   e2800002        add     r0, r0, #2
```

(ARM découpe son argument en **`0xa6=0x298/4`** – car **`pos=0x0f`** – suivi de **`2`** – car **`pos=0x00`** – afin de manipuler des arguments sur 32 bits par morceaux de 12 bits **[25]**. Si nous voulons renvoyer **`555`**, alors nous devons modifier les deux séquences d'opcodes, de **`e2800fa6`** en **`e2800f8a`** pour **`add r0, r0, #552`**, et **`e2800002`** en **`e2800003`** pour **`add r0, r0, #3`**). Les séquences d'opcodes qui nous intéressent sont donc dans le premier cas **`{0x8d,0x87,0x9a,0x02,0x00}`** et dans le second cas **`{0xe2,0x80,0x0f,0xa6,0xe2}`**.

Nous constatons au chargement puis déchargement de ce module les messages suivants par **`dmesg`** :

Terminal

```
[24254.596117] ffff880000000000
[24254.596120] fonction cherche @ ffffffffc09f2000
[24254.596121] code offset ffffffffc09f20a5
[24262.343786] sortie du module : 555
```

ORGANISATION DE LA MÉMOIRE CONTENANT LES INSTRUCTIONS

On notera que, alors que le tableau `s` qui contient la nouvelle séquence d'opcodes présente des données dans le même ordre que la sortie de `objdump` sur x86, nous avons inversé la séquence d'octets pour ARM. Bien que les deux architectures soient little endian – notre configuration de buildroot sélectionne `BR2_ENDIAN="LITTLE"` – et placent donc l'octet de poids le plus faible à l'adresse la plus faible (donc « à l'envers » si on lit de gauche à droite), la séquence des instructions dans une architecture CISC telle que Intel x86 (64) – instructions et arguments de taille variable – se lit octet par octet en incrémentant les adresses. Ainsi, les octets successifs que nous plaçons en mémoire suivent la séquence fournie par `objdump`. Au contraire, une architecture RISC comme ARM aligne nécessairement ses instructions (incluant l'argument) sur des multiples de 4-octets, et les instructions – de taille fixe de 4 octets – sont agencées elles aussi en little endian. On s'en convaincra avec le petit exemple ci-dessous (exécuté en utilisateur) qui renvoie d'une part l'organisation d'un mot de 16 bits en mémoire (on vérifie ainsi être sur des architectures little endian), puis les premiers opcodes de la fonction `main` :

Fichier

```
#include <stdio.h>
int main()
{short s=0x1234;
 char *c=(char*)&s;
 unsigned long offset=(unsigned long)&main,k;
 printf("%hhx %hhx\n",*c,*(c+1));
 for (k=0;k<16;k++) printf("%hhx ",*(char*)(offset+k));
 printf("\n");
}
```

Sur Intel x86 64, nous obtenons :

Terminal

```
34 12
55 48 89 e5 48 83 ec 20 66 c7 45 e6 34 12 48 8d
```

Tandis que `objdump -d endian.pc | grep -A5 \<main\>` nous dit :

Terminal

```
0000000000400576 <main>:
400576: 55 push %rbp
400577: 48 89 e5 mov %rsp,%rbp
40057a: 48 83 ec 20 sub $0x20,%rsp
40057e: 66 c7 45 e6 34 12 movw $0x1234,-0x1a(%rbp)
400584: 48 8d 45 e6 lea -0x1a(%rbp),%rax
```

Les séquences d'opcodes sont les mêmes, et nous constatons bien la taille variable des instructions et de leurs arguments, signature d'une architecture CISC. Au contraire sur processeur A13 (cœur Cortex A8 d'architecture ARM v7 [26]), nous observons :

Terminal

```
# ./endian.arm
34 12
0 48 2d e9 4 b0 8d e2 10 d0 4d e2 34 32 1 e3
```

qui propose une présentation en little endian – donc commençant par les arguments et finissant par l'instruction – des opcodes fournis par `objdump` sous forme d'instruction suivie des arguments (lecture naturelle pour un développeur) par :

Terminal

```
$ arm-buildroot-linux-uclibcgnueabihf-objdump -d endian.arm | grep -A4
\<main\>
000084ac <main>:
84ac: e92d4800 push {fp, lr}
84b0: e28db004 add fp, sp, #4
84b4: e24dd010 sub sp, sp, #16
84b8: e3013234 movw r3, #4660 ; 0x1234
```

Ceci prouve bien que les instructions de `adieu()` ont été modifiées pour renvoyer `555` au lieu de `666`.

3.6 Modifier le résultat d'une fonction du noyau

Cependant, ce cas est trivial, car le symbole recherché se trouve dans le même module que celui chargé de modifier le contenu de la mémoire. Nous n'avons pas réussi à balayer de façon systématique toute la mémoire du noyau, car nous nous heurtons à la protection mémoire de la MMU qui interdit d'accéder à des pages non allouées [5] : des outils tels que **fmem** [27] ou **LiME** [28][29] y parviennent, mais au prix de nombreux tests trop longs à expliciter ici.

Cependant, avons-nous réellement besoin de scanner toute la mémoire pour trouver le code associé à une fonction implémentée par un autre pilote ou par le noyau ? De la même façon que nous avions cherché la table des appels systèmes en partant de l'appel système `sys_close()`, nous allons limiter notre recherche à la zone mémoire que nous supposons contenir la fonction attaquée. Comme avec les appels systèmes, un point un peu délicat tient au fait que chaque module n'exporte pas ses symboles (notez que par ailleurs presque toutes les définitions de fonctions des modules du noyau sont préfixées de `static`, donc avec une portée locale au fichier source uniquement). Illustrons ce concept en modifiant le contenu de `/proc/cpuinfo` par modification de la zone mémoire appelée lors de l'affichage du contenu de ce pseudo-fichier :

1. dans les sources du noyau, nous constatons que `/proc/cpuinfo` est rempli par les fonctions de `linux-4.4.2/arch/x86/kernel/cpu/proc.c` et en particulier la fonction `show_cpu_info()` qui affiche la fréquence en MHz. Cependant, aucun symbole des fonctions de ce fichier source n'est exporté ;
2. nous constatons dans `/proc/kallsyms` que `show_cpu_info` se trouve en mémoire juste après `x86_match_cpu`, qui lui est un symbole exporté : cela est cohérent avec le fait que le code source de cette nouvelle fonction se trouve dans le même répertoire :

Terminal

```
# grep -A2 x86_match_cpu /proc/kallsyms | head -3
ffffffff8103e5d0 T x86_match_cpu
ffffffff8103e670 t c_stop
ffffffff8103e680 t show_cpuinfo
```

3. nous trouvons le prototype de `x86_match_cpu` dans `linux-4.4.2/arch/x86/include/asm/cpu_device_id.h` qui permettra donc la compilation de notre pilote qui recherchera l'emplacement en mémoire des fonctions que nous désirons manipuler,
4. finalement, partant d'un point de départ d'une page mémoire allouée au noyau, nous recherchons la séquence d'opcodes, ou dans notre cas la chaîne de caractères représentative de la manipulation à effectuer en mémoire, et écrasons le contenu de cette zone mémoire avec notre nouveau message.

Initialement, le fichier `/proc/cpuinfo` nous informe des performances de nos processeurs par les messages :

Terminal

```
# grep Hz /proc/cpuinfo
model name      : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
cpu MHz         : 1750.835
model name      : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
cpu MHz         : 1899.421
model name      : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
cpu MHz         : 1813.601
model name      : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
cpu MHz         : 1898.914
```

Nous chargeons le module dont le code source est le suivant :

Fichier

```
#include <linux/module.h>
#include <asm/cpu_device_id.h> // prototype de x86_match_cpu

long cherche(void)
{unsigned char sct,sctp1,sctp2,sctp3,sctp4;
 unsigned long int offset = (unsigned long)&x86_match_cpu; // fonction
proche de l'appel qu'on va modifier
 printk(KERN_INFO "%lx\n",offset);
 while (offset < ULLONG_MAX) // recherche dans la mémoire allouée au noyau
   {sct  = *(unsigned char*)offset;
    sctp1= *(unsigned char*)(offset+1);
    sctp2= *(unsigned char*)(offset+2);
    sctp3= *(unsigned char*)(offset+3);
    sctp4= *(unsigned char*)(offset+4);
 if ((sct=='c') && (sctp1=='p') && (sctp2=='u') && (sctp3==' ') &&
(sctp4=='M'))
       {printk(KERN_INFO "code offset %lx\n",(unsigned long)offset);
        printk(KERN_INFO "... %hhx %hhx %hhx\n",(unsigned char)
sctp4,*(unsigned char*)(offset+5),*(unsigned char*)(offset+6));
        return(offset);
       }
    offset++;
   }
 return offset;
}

static int __init module_start(void)
{unsigned long offset;
 unsigned long original_cr0;
 char s[16]="toto THz\t: 42 \0";  // nouveau message à afficher
 offset=cherche();
 original_cr0 = read_cr0();    // passe la page des appels sys en écriture
write_cr0(original_cr0 & ~0x00010000);
 memcpy((char*)offset,s,16);
 write_cr0(original_cr0);    // passe la page des appels en lecture seule
 return(0);
}

static void __exit module_end(void) {printk(KERN_INFO "sortie du module");}

module_init(module_start);
module_exit(module_end);
MODULE_LICENSE(«GPL»);
```

Pour désormais obtenir :

Terminal
```
# insmod 3mymod.ko
# grep Hz /proc/cpuinfo
model name      : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
toto THz        : 42  cache size        : 3072 KB
model name      : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
toto THz        : 42  cache size        : 3072 KB
model name      : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
toto THz        : 42  cache size        : 3072 KB
model name      : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
toto THz        : 42  cache size        : 3072 KB
```

Nous voici donc en possession d'un ordinateur qui ne fonctionne plus sur des « cpu », mais sur des « toto », cadencés à `42 THz`. On prendra soin de ne pas recharger une seconde fois ce pilote, puisque maintenant la chaîne recherchée « cpu M » n'existe plus dans `/proc/cpuinfo` et le balayage de la mémoire heurtera immanquablement une page non allouée au noyau, se traduisant par un accès illégal et un pilote qui ne pourra plus être détaché du noyau.

Au chargement du pilote, `dmesg` nous informe des adresses des divers emplacements qui nous intéressent en commençant par l'emplacement de la fonction `x86_match_cpu` puis de la chaîne de caractères recherchée :

Terminal
```
[  175.403604] ffffffff8103e5d0
[  175.416773] code offset ffffffff817cd383
[  175.416777] ... 4d 48 7a
```

Ces informations sont cohérentes une fois de plus avec `/proc/kallsyms`. Nous avons donc démontré notre capacité à intervenir sur la zone mémoire allouée à une partie du noyau autre que notre propre module. Le passage de la manipulation des chaînes de caractères affichées au contenu de la mémoire (variables) ou instructions n'est plus qu'une question d'analyse du code désassemblé du noyau pour identifier la séquence d'opcodes ou l'emplacement de la variable à modifier.

Cette approche a par exemple été utilisée, il y a maintenant fort longtemps, par l'auteur pour débloquer quelques fonctions du logiciel propriétaire **Framework** de contrôle de potentiostat PC3-300 de **Gamry** : accéder à certaines fonctionnalités se traduisait par un message d'erreur. Une fois le logiciel désassemblé, l'adresse du message d'erreur est recherchée, puis l'instruction qui prend en argument cette adresse. Nous pouvons supposer que le bout de code qui fait appel au message d'erreur est aussi celui chargé d'autoriser l'accès à ces fonctionnalités. La condition donnant l'autorisation étant identifiée, il suffit de modifier l'opcode de saut avec une condition (`je` : *jump if equal*) par son opposé (`jne` : *jump if not equal*) pour débloquer la fonctionnalité du logiciel. Cette approche est d'autant plus valide à l'époque actuelle où de nombreux instruments scientifiques sont commercialisés avec toutes les fonctions matérielles installées et le fabricant demande un paiement additionnel pour simplement débloquer la fonctionnalité logicielle.

CONCLUSION

Nous avons présenté quelques méthodes de modification de la mémoire d'un système informatique exécutant GNU/Linux en insérant un module chargé soit de trouver les appels systèmes et rediriger les appels vers nos propres fonctions, soit d'écraser les fonctions

d'origine pour les remplacer avec nos propres instructions. Au-delà de la capacité à modifier le fonctionnement de systèmes fournis uniquement sous forme de binaire, est-ce que ces modifications peuvent importer au commun des utilisateurs ? Il nous semble qu'avec la prolifération des systèmes embarqués sous GNU/Linux (box de connexion internet, récepteurs GPS, disques durs) avec l'absence du respect des concepts de base de sécurité (absence de compte utilisateur avec seule une connexion *root*, mots de passe en dur dans l'image flashée sur le système embarqué [30][31]), une compréhension des méthodes de corruption du noyau peut être utile. La mode des objets connectés, ou IoT, va probablement encore empirer notre exposition aux attaques en éliminant la couche physique protégeant l'accès au périphérique – les liaisons radiofréquences étant accessibles à tout interlocuteur à proximité du périphérique – et il n'y a aucun doute que les attaques se multiplieront dans cette direction, avec la capacité à cacher ses traces selon certains des mécanismes discutés ici. Y a-t-il une perspective d'amélioration de la protection contre ces attaques ? Nous plaçant au plus bas niveau du système d'exploitation, où seule la MMU (matériel) handicape nos capacités d'action, il est peu probable qu'une solution stable existe. Les diverses versions de Linux pallient à quelques méthodes disponibles sur Internet qu'un utilisateur qui se contente de copier sans comprendre ne pourra réappliquer, mais nous avons vu ici toutes les étapes pour reprendre étape par étape la démarche d'identification de la zone mémoire à modifier, information qui est nécessairement accessible si le système d'exploitation doit pouvoir fournir le service pour lequel il est conçu. [32] discute de la sécurité amenée par la redistribution aléatoire des accès mémoire, qui rompt au moins partiellement l'hypothèse de proximité des appels systèmes ou des structures de données exportées par rapport aux emplacements recherchés. La principale limitation citée par [33] est la nécessité de reproduire un environnement local dupliquant la configuration du noyau attaqué sur le système distant : tant qu'un noyau de **kernel.org** est utilisé, l'obtention de la version (`uname -a`) et de la configuration (`/boot/config*`) devraient rendre cette étape faisable. ■

RÉFÉRENCES ET NOTES

[1] M.H. Ligh, A. Case, J. Levy & A. Walters, *« The Art of Memory Forensics – Detecting Malware and Threats in Windows, Linux, and Mac Memory »*, Wiley, 2014

[2] D.J. Barrett, R.E. Silverman & R.G. Byrnes, *« Linux Security Cookbook »*, O'Reilly, 2003

[3] R.J. Hontanon, *« Linux Security »*, Sybex, 2001

[4] R. O'Neill, *« Learning Linux Binary Analysis »*, Packt Publishing, 2016

[5] madsys, *« Finding hidden kernel modules (the extrem way) »*, Phrack 61, 2003

[6] *« Malware hits millions of Android phones »* sur : **http://www.bbc.com/news/technology-36744925**

[7] D.-H You, *« Android platform based linux kernel rootkit »*, Phrack 68, 2011, et *« 6th IEEE International Conference on Malicious and Unwanted Software (MALWARE) »*, 2011

[8] Buildroot : **https://github.com/trabucayre/buildroot**

[9] Les cas où l'architecture du processeur induit des différences de code source sont gérés en testant la constante `__ARMEL__` indicatrice d'une cross-compilation à destination de ARM, tel que nous en informe `arm-buildroot-linux-uclibcgnueabihf-gcc -dM -E - < /dev/null | grep ARM`

[10] Silvio Cesare, *« Unix viruses »* sur : **https://www.win.tue.nl/~aeb/linux/hh/virus/unix-viruses.txt**

[11] B. Hatch, J. Lee & G. Kurtz, *« Hacking Linux exposed: Linux security secrets & solutions »*, Osborne/McGraw-Hill, 2001

[12] sd & devik, *« Linux on-the-fly kernel patching without LKM »*, Phrack 58, 2001

[13] Norme POSIX : **http://pubs.opengroup.org/onlinepubs/009695399/idx/functions.html**

[14] Ce point a rapidement été abordé, naïvement, en annexe C de la thèse de l'auteur disponible sur : **https://hal.archives-ouvertes.fr/tel-00509641/document** dans le contexte d'un coprocesseur Z80 sur carte ISA de PC

[15] **http://www.gilgalab.com.br/hacking/programming/linux/2013/01/11/Hooking-Linux-3-syscalls/**

[16] Assembleur AVR : **http://www.atmel.com/webdoc/avrassembler/avrassembler.wb_ICALL.html**

[17] **https://bbs.archlinux.org/viewtopic.php?id=139406**

[18] **turbochaos.blogspot.fr/2013/09/linux-rootkits-101-1-of-3.html** ou **gadgetweb.de/linux/40-how-to-hijacking-the-syscall-table-on**

[19] **https://poppopret.org/2013/01/07/suterusu-rootkit-inline-kernel-function-hooking-on-x86-and-arm/**

[20] Control register : **https://en.wikipedia.org/wiki/Control_register**

[21] M.A. Ludwig, *« The Little Black Book of Computer Viruses – Volume 1: the basic technology »*, American Eagle Publications, 1990, disponible sur **http://www.cin.ufpe.br/~mwsa/arquivos/THE_LITTLE_BLACK__BOOK_OF_C.PDF** – le lecteur se souviendra peut-être qu'à la sortie de cet ouvrage – *« Naissance d'un virus »* (Addison & Wesley) en 1993, nombre de clients – y compris l'auteur – accompagnaient l'achat de cet ouvrage d'une introduction à l'assembleur x86, avec rupture de stock de *« L'assembleur facile »* aux éditions Marabout (1989) !

[22] M.A. Ludwig, *« The Little Black Book of Computer Viruses – Computer Viruses, Artificial Life and Evolution »*, 1993

[23] M.A. Ludwig, *« The Giant Black Book of Computer Viruses »*, 1995

[24] A. Lineberry, *« Malicious Code Injection via /dev/mem »*, BlackHat Europe, 2009

[25] **http://www.peter-cockerell.net/aalp/html/ch-3.html**

[26] ARMv7-M Architecture Reference Manual, 2010, sur **http://www.pjrc.com/teensy/beta/DDI0403D_arm_architecture_v7m_reference_manual.pdf**, l'accès à la version officielle sur **http://infocenter.arm.com** nécessitant de s'enregistrer : *« section A3.3.1 Control of endianness in ARMv7-M »*, nous apprenons que « The endianness setting only applies to data accesses. Instruction fetches are always little endian. »

[27] P. Wächter & M. Gruhn, *« Practicability study of Android volatile memory forensic research »*, IEEE Workshop on Information Forensics and Security (WIFS), 2015

[28] **http://hysteria.cz/niekt0/**

[29] **https://github.com/504ensicsLabs/LiME**

[30] C. Heffner, *« Exploiting Network Surveillance Cameras Like a Hollywood Hacker »*, Black Hat, 2013, disponible sur : **https://www.youtube.com/watch?v=B8DjTcANBx0**

[31] GTVHacker, *« Hack All The Things: 20 Devices in 45 Minutes »*, Defcon 22, 2014, disponible sur **https://www.youtube.com/watch?v=h5PRvBpLuJs**

[32] D.M Stanley, D. Xu & E.H. Spafford, *« Improved kernel security through memory layout randomization »*, IEEE 32nd International Performance Computing and Communications Conference (IPCCC), 2013

[33] J. Stüttgen, & M. Cohen, *« Robust Linux memory acquisition with minimal target impact »*, Digital Investigation 11, 2014, pp.S112–S119 sur : **http://www.sciencedirect.com/science/article/pii/S174228761400019X**

REMERCIEMENTS

Je remercie S. Guinot (association Sequanux, Besançon) et F. Tronel (CentraleSupélec/Inria, Rennes) pour avoir répondu à mes questions et orienté mes recherches au cours de cette étude.

CAS N°5

UN CAS CONCRET : CORRECTION DU PILOTE HITACHI STARBOARD EN 64 BITS

François REVOL

De nombreux fabricants de périphériques publient désormais des pilotes plus ou moins libres pour Linux, à défaut des spécifications (qui pourtant sont le « manuel utilisateur » pour nous développeurs), mais tous n'assurent pas forcément le suivi adéquat.

Le tableau blanc interactif (TBI) Hitachi StarBoard Link EZ2 [1] est fourni avec une version « Linux » du logiciel propriétaire dédié (à condition de fouiller un peu), et même d'un pilote fourni avec le code source, mais qui malheureusement n'est utilisable « que sur Linux 32 bits » d'après la documentation. Il a donc été acheté sans crainte par le patron d'une PME locale utilisant GNU/Linux. Croyant donc acheter un produit utilisable sous Linux, on se retrouve au final avec un objet inutilisable, faisant même planter joyeusement le noyau 64 bits avec le seul patch non officiel disponible [2] lorsqu'après force bidouille le pilote compile enfin.

La capitulation étant bien sûr hors de question, essayons donc de comprendre et corriger le problème.

1. Ô KERNEL, SUSPEND TON VOL, ET VOUS, VERSIONS D'API…

Linux n'a pas d'API stable pour les pilotes de périphériques, tant pour raison technique – ne pas freiner l'évolution du noyau – qu'à des fins « politiques », pour obliger les auteurs de pilotes à publier leur code et le maintenir au sein même du dépôt officiel. Si l'on peut comprendre la motivation technique, bien que d'autres noyaux même libres évoluent très bien malgré une API stable, l'action politique est loin d'avoir fait mouche lorsque l'on regarde les *glues* spaghetti des pilotes binaires **Nvidia** ou celles d'**OSS4**. Mais bon, il faut faire avec. Cela a néanmoins plusieurs conséquences :

- plutôt que de s'inscrire dans un cadre officiel dont on ne sait pas comment il évoluera, ou d'oser en proposer un [3], certains codent dans leur coin, avec des `ioctl()` voire des modules spécifiques pour tout ce qui n'est pas strictement requis, notamment pour communiquer avec un démon spécial en espace utilisateur ;
- même lorsque le code du pilote est disponible, il est rarement proposé pour inclusion dans les sources officielles du noyau, surtout lorsqu'il n'est qu'une partie nécessitant un démon propriétaire, donc aussi peu maintenable (et donc maintenu) que des pilotes binaires utilisant le tas de `#ifdef` déjà évoqué.

DÉFINITION

L'appel système `ioctl()` (pour « I/O control ») est une sorte de fourre-tout, permettant de nombreuses interactions avec les pilotes hors des fonctions habituelles applicables sur les descripteurs de fichier (`write`, `read`, `mmap`, etc.). Apparu dans la version 7 de l'Unix d'AT&T, il n'est défini dans POSIX [4] que pour l'extension obsolète `STREAMS`, avec un nombre d'arguments variable.

Il prend en paramètres un descripteur de fichier, un code définissant la requête invoquée, et en option un argument (souvent, mais pas toujours, un pointeur sur un tampon spécifique à la requête). Certains OS demandent la taille du tampon en quatrième argument. Sous Linux, le code de la requête est construit par les macros `_IO()` et ses variantes déclarées dans `/usr/include/asm-generic/ioctl.h`, en incluant l'indication de direction et de taille du tampon passé.

Suivant l'OS il est aussi utilisé pour *prototyper* de nouvelles fonctions sans ajouter d'appel système. Par exemple sous BeOS, la pile réseau BONE qui ajoutait des *sockets* BSD en utilisant de vrais descripteurs de fichiers vers un `/dev/net/api` implémentait les appels tels que `sendmsg()` par des requêtes `ioctl()`.

C'est de plus une source potentielle de failles de sécurité, car il incombe au pilote de valider les adresses passées depuis l'espace utilisateur, puisque la requête lui est spécifique et le noyau ne sait donc pas l'analyser.

2. FONCTIONNEMENT D'UN TBI

Un TBI, également appelé *Tableau Numérique Interactif* (TNI), est habituellement composé de :

- une surface blanche, en général un tableau pour marqueurs effaçables ;
- un vidéoprojecteur relié à un ordinateur ;
- un dispositif d'acquisition du pointage.

Pour le StarBoard :

- un bloc de capteurs connecté par USB et maintenu en haut du tableau par des aimants, constitué de petites caméras pour analyser l'ombre portée par le doigt ou le stylo éclairé par plusieurs LEDs infra-rouges ;
- des barres réfléchissantes à positionner sur les bords du tableau.

Le pilote d'un tel matériel, qui équivaut à une grosse tablette graphique, doit donc générer des événements de pointage absolu à partir des images brutes récupérées par les caméras. Le traitement d'image n'étant pas vraiment une tâche aisée en mode noyau, il est souvent délégué à un démon utilisateur, qui renvoie au pilote les événements à faire suivre au sous-système d'entrée du noyau.

3. LE CAS STARBOARD

Concernant le pilote StarBoard, l'utilisation de codes `ioctl()` spécifiques est effectivement due à l'architecture du pilote constitué de deux parties :

- `lsadrv.ko` : c'est ce module qui dialogue directement avec le périphérique USB, et qui fait suivre les données au service puis publie des événements de pointage, et c'est en premier lieu celui-ci qui pose souci, et dont les sources sont heureusement disponibles.
- le service « `starboard` » (au sens Unix) qui en fait lance plusieurs démons binaires 32bits : `lsadrv` et `DGBoard`, le premier étant probablement responsable de la détection des mouvements.

D'autres outils binaires servant notamment à la calibration, mais pas vraiment documentés sont disponibles, et un module optionnel `coach.ko` sert de pilote à une webcam sur bras articulé pour l'affichage de documents papier. Le reste des 280 Mio constituent le logiciel StarBoard lui-même.

Lors de l'installation du paquet, un script vérifie la version du noyau et copie une version précompilée du module s'il en trouve une, ou à défaut tente de le compiler à partir des sources incluses. C'est bien entendu là que l'installation échoue sur un noyau récent pour la dernière version connue du paquet (lien cassé depuis) trouvée au détour d'un forum [5].

NOTE

Malheureusement, l'installation sur une Debian récente n'a pas abouti à cause de dépendances obsolètes, les tests ont donc été faits dans une VM Ubuntu 14.04. Il reste à espérer qu'Hitachi publie des binaires plus à jour.

4. CORRECTION DU PILOTE

Après quelques tentatives infructueuses d'utilisation des *patches* existants, incomplets et « plantogènes », il faut prendre les choses en main. En premier lieu par la création d'un dépôt Git et d'un projet GitHub [6] pour garder l'historique des tentatives, puis l'étude du code.

4.1 Modifications dues à procfs

Le premier problème vient de la suppression en version 3.10 de `create_proc_entry()`, déjà dépréciée depuis longtemps. Le code publiant la liste des matériels détectés dans `/proc/driver/lsadrv/devices` doit utiliser la nouvelle API. Le premier des *patches* trouvés tente d'utiliser `proc_create()`, mais réutilise la fonction `lsadrv_read_devices()` sans tenir compte de la différence de signature. Le noyau l'appelle donc avec des paramètres incorrects, ce qui se solde irrémédiablement par un joli *oops*.

Cette fonction est appelée pour énumérer les périphériques détectés et supportés par le pilote, et doit donc remplir le *buffer* avec une chaîne de caractères formatée, une ligne par identifiant. Plutôt que de tenter de corriger le code existant en ajoutant des `#ifdef` partout, nous pouvons réécrire cette partie en utilisant l'API `seq_file` spécialement créée pour simplifier cette tâche. Plutôt qu'un `sprintf()` qui ne vérifiait même pas la taille du *buffer* disponible suivi d'une boucle pour calculer la taille écrite, on appelle alors `seq_printf()` qui se débrouille tout seul. Le code est alors bien plus lisible :

Fichier

```
#include <linux/seq_file.h>

static int lsadrv_devices_show(struct seq_file *m, void *v)
{
 struct list_head *tmp;
 down(&device_list_lock);
 tmp = device_list.next;
 while (tmp != &device_list) {
  struct lsadrv_device *xdev = list_entry(tmp, struct lsadrv_device, device_list);
  tmp = tmp->next;
  seq_printf(m, "%03d/%03d\n", xdev->udev->bus->busnum, xdev->udev->devnum);
 }
 up(&device_list_lock);
 return 0;
}
```

On indique au noyau d'appeler notre fonction pour lire lors de l'ouverture du fichier :

Fichier

```
static int lsadrv_devices_open(struct inode *inode, struct file *file)
{
 /* should use seq_open() but this should be sufficient for the need */
 return single_open(file, lsadrv_devices_show, NULL);
}
```

Des cas plus compliqués utiliseront `seq_open()`. Nous passons la main pour le reste :

Fichier

```
static struct file_operations proc_fops = {
   .open     = lsadrv_devices_open,
   .read     = seq_read,
   .llseek   = seq_lseek,
   .release  = seq_release
};
```

4.2 Mutex or not mutex

Le second patch trouvé corrige les appels d'initialisation des mutex (verrou d'exclusion mutuelle) protégeant le pilote des accès concurrents. En effet, les macros utilisées pour leur initialisation ont été supprimées. Il suffit donc de remplacer l'appel à `init_MUTEX()` par un appel direct à `sema_init()`.

4.3 Appels ioctl() 32bits

À ce stade, le pilote se charge et détecte correctement le matériel. Malgré tout, le logiciel StarBoard ne trouve pas le tableau. Quelques `printk` bien placés en montrent la cause : les codes `ioctl()` utilisés par les démons compilés en 32bit sont différents de ceux reconnus par le pilote.

Nous l'avons dit, Linux code la taille de la structure passée dans la valeur de la requête avec les macros `_IO()`. Or la structure utilisée pour échanger avec le pilote contient des pointeurs, dont la taille change en 64 bits, sans parler de l'alignement des membres dans les structures. Le programme d'exemple `decode_ioctl.c` (dans le dépôt Git) compilé deux fois (avec et sans `CFLAGS=-m32`) le montre facilement :

Terminal

```
$ make decode_ioctl && ./decode_ioctl CFLAGS=-m32 0x401078c8
...
DIR:  1 0x00000001
TYPE: 120 0x00000078 'x'
NR:   200 0x000000c8
SIZE: 16 0x00000010
...
sizeof(struct lsadrv_bulk_transfer_control) = 16
...
$ rm decode_ioctl && make decode_ioctl && ./decode_ioctl 0x401078c8
...
sizeof(struct lsadrv_bulk_transfer_control) = 24
...
```

Donc lorsque le code utilisateur appelle `ioctl(..., 0x401078c8, ...)` avec un pointeur sur un `struct lsadrv_bulk_transfer_control` de 16 octets, notre pilote compilé lui en 64 bits attend joyeusement une valeur différente puisque calculée avec une structure dont la taille est passée à 24 octets !

Depuis le noyau 2.6.11 **[7]**, plusieurs possibilités sont offertes aux pilotes pour implémenter ioctl :

- le `ioctl` originel, qui est déprécié ;
- `unlocked_ioctl` qui est une version appelée sans détenir le **BKL** (*Big Kernel Lock*), l'équivalent du *Giant Lock* de **FreeBSD**, c'est-à-dire le verrou global qui suspend la préemption des *threads* du noyau. Le BKL est une réminiscence de l'époque où Linux comme la plupart des noyaux n'était pas lui-même préemptible, c'est-à-dire que les appels système ne pouvaient être interrompus qu'en certains endroits bien précis. Depuis il y a eu le patch PREEMPT et bien d'autres changements, rendus nécessaires par le nombre croissant de cœurs dans les processeurs. Le but étant bien sûr de limiter l'utilisation du BKL au strict minimum. Un post sur la LKML **[8]** indique comment modifier son `ioctl` en `unlocked_ioctl` en rajoutant les appels à `[un]lock_kernel()` au besoin. A priori, notre pilote n'a pas besoin du BKL ;
- `compat_ioctl`, qui est appelé préférentiellement (sans le BKL) depuis les processus 32 bits, permet de corriger les arguments avant d'effectuer le même traitement que la version `unlocked`. S'il indique ne pas supporter un code (`ENOIOCTLCMD`) ou est absent, le noyau tentera quelques conversions connues avant d'appeler `unlocked_ioctl`, ou `ioctl` en dernier ressort après avoir acquis le BKL.

On peut tester les macros `CONFIG_COMPAT` pour le support 32 bits sur 64 bits, ainsi que `HAVE_UNLOCKED_IOCTL` et `HAVE_COMPAT_IOCTL`.

Malheureusement pour nous la partie du pilote concernée n'est pas appelée directement par le VFS, mais par la pile USB, qui ne permet pas d'indiquer un **compat_ioctl**, donc nous nous contenterons de rajouter des **case** différents dans la version *unlocked*, calculés à partir de structures redéclarées avec des types appropriés, comme **compat_caddr_t** pour les pointeurs. Par exemple, pour les transferts *bulk* :

Fichier

```
#include <linux/compat.h>  /* for 32bit compatibility */

#ifdef CONFIG_COMPAT
struct compat_lsadrv_bulk_transfer_control
{
 unsigned int ep;
 unsigned int len;
 unsigned int timeout; /* in milliseconds */
 compat_caddr_t data; /* (void *) */
} __attribute__ ((packed));
```

Le **packed** permet d'éviter que le compilateur n'aligne la structure avec les conventions 64 bits.

Fichier

```
#define LSADRV_IOC_BULK32 _IOW(LSADRV_IOC_MAGIC, LSADRV_IOCTL_BASE + 9,
struct compat_lsadrv_bulk_transfer_control)
```

Dans notre ioctl, on déclare une variable **karg** qui pointera sur les arguments corrigés, et on rajoute quelques cas :

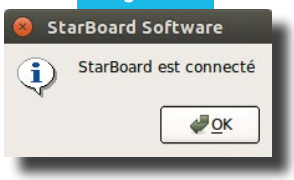
Fichier

```
int lsadrv_usb_ioctl(struct lsadrv_device *xdev, unsigned int cmd, void
*arg)
{
 int ret = 0;
 void *karg = NULL;
... switch (cmd) { ...
#ifdef CONFIG_COMPAT
  /* no need for get_user/put_user here */
  case LSADRV_IOC_BULK32:
  {
   struct compat_lsadrv_bulk_transfer_control *ua32 = arg;
   struct lsadrv_bulk_transfer_control *a;

   a = karg = kmalloc(sizeof(*a), GFP_KERNEL);
   if (!karg) return -ENOMEM;

   a->ep = ua32->ep;
   a->len = ua32->len;
   a->timeout = ua32->timeout;
   a->data = compat_ptr(ua32->data);

   ret = lsadrv_ioctl_bulk(xdev, a);
   break;
  }
  ...
 }
 ...
 kfree(karg);
```

Notez que la pile USB nous a déjà copié le *buffer* dans le noyau, nous épargnant l'utilisation de `get_user()`, mais le cas général en a besoin. On copie les membres un par un, notez l'appel à `compat_ptr` pour le pointeur, puis on appelle la fonction de transfert idoine. On terminera bien sûr en libérant `karg`.

On recompile... ouéééé ! On testouille... ça devrait ressembler à la figure 1... Youpi, ça marche \o/

Figure 1



Le TBI est enfin détecté !

5. LES LOGICIELS DE TBI

Faute d'expérience sur leur utilisation, nous nous contenterons de mentionner les caractéristiques des logiciels pouvant utiliser ce matériel.

5.1 StarBoard

StarBoard est le logiciel publié par Hitachi spécifiquement pour leur matériel. La figure 2 montre l'interface principale. S'il semble assez complet, il est propriétaire et spécifique à leur hardware (il refusera de fonctionner s'il ne détecte pas l'ID USB du tableau). Il a au moins le mérite d'être disponible pour GNU/Linux (enfin, en binaire 32 bits), ce qui n'est pas forcément le cas de la concurrence.

Figure 2

Test du TBI dans une VM Ubuntu 64bit avec le logiciel StarBoard.

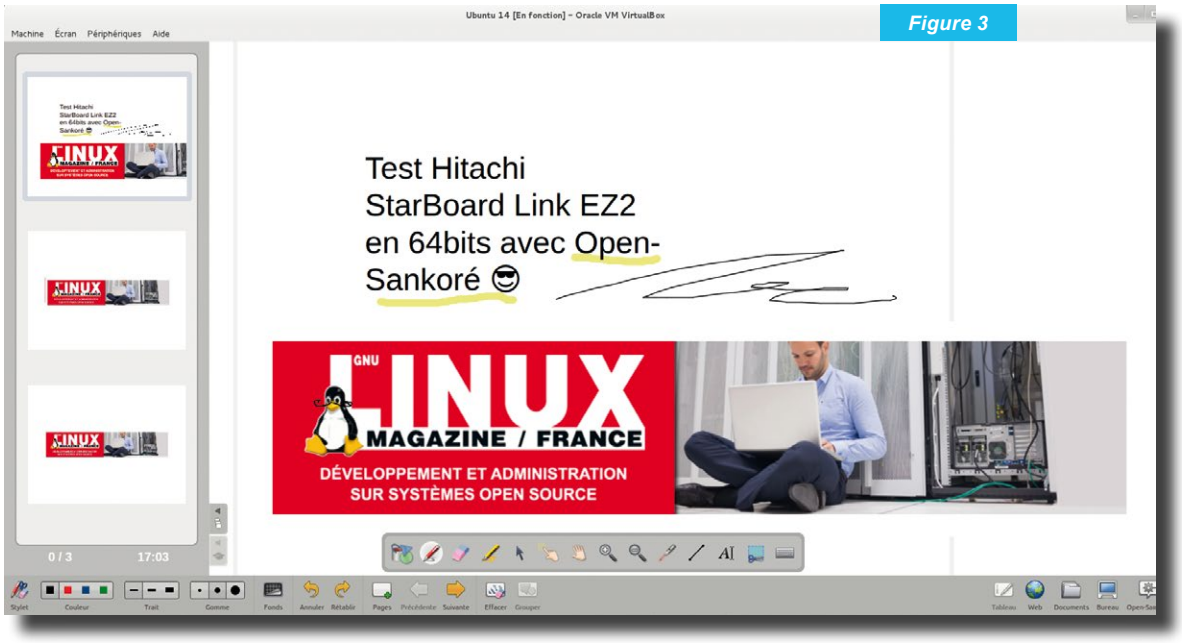
5.2 OpenSankoré

À l'opposé, **OpenSankoré [9]** est libre, et fonctionne avec tous les TBI disponibles sur le marché, les PC et les tablettes, ce qui permet à un enseignant de préparer son cours chez lui sans nécessiter le matériel de la classe.

Son interface graphique est épurée (figure 3), mais au final plus intuitive et même plus réactive dans une VM que StarBoard.

OpenSankoré, ou plutôt son *fork* helvète plus actif **OpenBoard [10]**, cherche d'ailleurs des volontaires pour maintenir des paquets Debian !

Figure 3



Test du TBI avec OpenSankoré dans la même VM.

CONCLUSION

Le support de GNU/Linux par les fabricants de périphériques ne va pas encore de soi. Lorsqu'ils arrivent à publier des pilotes (alors qu'on leur demande seulement les spécifications), ils considèrent souvent qu'il suffit de déposer le tout dans un coin, sans tenir compte de la diversité des distributions et de la rapidité d'évolution d'un OS libre.

Au moins un code source peut se *patcher*, et nous avons vu comment. Quant à espérer que le fabricant intègre les *patches*... je n'ai toujours pas de réponse d'Hitachi, qui d'ailleurs vient de revendre la marque *StarBoard* **[11]**. ■

RÉFÉRENCES

[1] Tableaux Blancs Interactifs *Hitachi StarBoard* : **http://psg.hitachi-solutions.com/starboard**

[2] *Patches* incomplets : **http://packages.altlinux.org/en/Sisyphus/srpms/kernel-modules-lsadrv-std-pae**

[3] LWN : *The platform problem* : **https://lwn.net/Articles/443531/**

[4] `ioctl()` dans POSIX : **http://pubs.opengroup.org/onlinepubs/9699919799/functions/ioctl.html**

[5] Forum Ubuntu : « Tableau Numérique Starboard : ça existe sur Linux ? » : **http://forum.ubuntu-fr.org/viewtopic.php?id=416758**

[6] Pilote StarBoard corrigé sur GitHub : **https://github.com/mmuman/starboard-lsadrv**

[7] LWN : *The new way of ioctl()* : **http://lwn.net/Articles/119652/**

[8] LKML : *Switch ioctl functions to ->unlocked_ioctl* : **https://lkml.org/lkml/2008/1/8/213**

[9] Logiciel libre pour TBI *OpenSankoré* : **http://open-sankore.org/fr**

[10] Le fork *OpenBoard* : **http://openboard.ch/**

[11] Nouveau site *StarBoard Solution* : **http://www.starboard-solution.com/**

3

3 VIRTUALISEZ

À découvrir dans cette partie...

3.1 KVM : focus sur l'implémentation d'un hyperviseur dans Linux

KVM est une implémentation open source assez récente (intégrée au noyau mainline 2.6.20, en 2007) d'un hyperviseur. Nous décrivons dans cet article son architecture logicielle, intégrant divers intervenants (module KVM, API Virtio, Qemu, la libvirt, etc.) pour mieux distinguer quel rôle a chacun de ces composants et comment ces derniers interagissent. Cet article décrit également la manière dont KVM utilise les extensions de virtualisation matérielles sur l'architecture Intel x86 et sur ARM ainsi que les contraintes associées. p. 88

KVM : FOCUS SUR L'IMPLÉMENTATION D'UN HYPERVISEUR DANS LINUX

Philippe THIERRY

Cet article décrit l'implémentation noyau de KVM, ses interactions avec le matériel et les contraintes associées sur les architectures Intel et ARM. Il a également pour but de décrire ce qui n'est pas du ressort du module, et quels rôles ont réellement Qemu, le noyau et la libvirt dans l'exécution d'une machine virtuelle.

Cet article a pour but de décrire comment est implémenté **KVM**, ses interactions avec le matériel avec l'écosystème logiciel de l'équipement. Le but est de montrer les concepts derrière l'implémentation de KVM à la fois sur x86, mais également sur les cibles embarquées. Les éléments de gestion ne sont pas traités, au profit des couches basses du logiciel et des API associées.

1. ARCHITECTURE MACROSCOPIQUE DE KVM

Pour commencer, je vous propose quelques définitions de termes employés dans cet article :

- KVM : *Kernel-based virtual machine*. KVM est une implémentation logicielle d'un hyperviseur *bare-metal* (classiquement nommé « type 1 » [1][2]) ;
- Qemu : Qemu est un logiciel d'émulation (Qemu pour *Quick Emulator*), en charge d'émuler une machine physique (séquence de boot, périphériques, etc.) ;
- hyperviseur, VMM : un hyperviseur (ou VMM, *Virtual Machine Monitor*) est une fonction logicielle et/ou matérielle en charge de créer et d'exécuter des machines virtuelles ;
- U-boot : U-Boot est un *bootloader* Open source, implémenté par **DENX** software. Il se substitue, dans les architectures embarquées (ARM, PowerPC, etc.) à une partie du BIOS (qui n'existe pas dans ces dernières) et à **GRUB** ;
- *Full-virtualization* : virtualisation complète, permettant d'exécuter un environnement logiciel complet sans modification. Peut se faire de deux manières :
  - à l'aide du matériel, on parle alors de HVM (*Hardware-assisted Virtualization*). Ce dernier est alors apte à différencier un contexte virtuel d'un contexte natif, et apporte des abstractions au niveau du matériel pour permettre l'exécution d'un système d'exploitation sans modification de ses instructions ;
  - sans aide du matériel. Cas historique, plus difficile à réaliser et coûteux (utilisation de mécanismes de « *binary translation* » pour détecter toute utilisation d'instructions invalides en avance de phase). L'arrivée du HVM a permis de fortement simplifier l'exécution non modifiée d'environnements logiciels ;
- para-virtualisation : virtualisation avec l'aide du logiciel virtualisé. Ce dernier est modifié pour faire des requêtes explicites à l'hyperviseur à certains moments de son exécution. Diverses fonctions peuvent être para-virtualisées :
  - le processeur (la MMU, accès à divers registres d'états, etc.) ;
  - les périphériques (exemple de **VirtIO**, que nous verrons plus loin).
- OS invité : système d'exploitation exécuté dans une machine virtuelle ;
- OS hôte : système d'exploitation, lorsqu'il existe, s'exécutant en dehors du contexte virtuel.

1.1 Le module KVM

Le module KVM est le composant logiciel en charge de gérer les fonctions matérielles de gestion de la virtualisation ainsi que les interactions avec les composantes d'hypervision du

noyau Linux dont il n'est pas le gestionnaire (comme l'ordonnancement des machines virtuelles). Selon les architectures matérielles, il peut être le seul composant logiciel à s'exécuter avec les droits d'accès aux API de virtualisation matérielle (c'est le cas sur ARM par exemple). Sa volumétrie de code est relativement petite au regard du noyau Linux dans son ensemble (de l'ordre de 50 000 lignes de code pour le support de l'ensemble des extensions x86, de l'ordre de 15 000 pour le support ARM sur un noyau 4.7.0).

1.2 Le rôle de Qemu

Qemu est un émulateur... et continue de l'être y compris dans le cadre de KVM. Son but est d'émuler un certain nombre de fonctions que le module KVM n'est pas capable de gérer par lui-même :

- la séquence de boot du matériel virtualisé : exécution d'un BIOS ou d'un X-loader, nécessaire pour pouvoir transmettre un certain nombre de structures de données en mémoire au bootloader (Grub dans le cas x86, U-boot sur ARM) ;
- l'émulation potentielle de divers périphériques (contrôleurs Ethernet, contrôleurs de disque et ainsi de suite) ;
- les communications inter-VMs ou, avec le noyau de l'OS hôte ou avec des périphériques physiques *remappés*, via des *threads* d'I/O, en utilisant des IPC standards.

Globalement, Qemu gère donc les séquences de démarrage/extinction et les I/O lorsque celles-ci ne correspondent pas à des accès directs à un périphérique physique ou *virtualisable* matériellement. Bien que la présence du processus `qemu` au-dessus du noyau de l'hôte donne l'impression que le noyau de la machine virtuelle s'exécute dans un contexte applicatif, il s'agit d'un faux-ami. Qemu peut être vu comme un élément à positionner comme une fonction d'émulation de certaines I/O positionnées à côté du contexte effectif de la machine virtuelle. Il se comporte alors comme une fonction de proxy d'I/O pour l'OS invité.

Il faut bien voir que Qemu n'est pas strictement interconnecté à KVM. Le module KVM fournit une API permettant à toute solution applicative apte à émuler le démarrage d'une machine physique et à *proxifier* des entrées-sorties de venir se positionner en lieu et place de Qemu.

1.3 Et la libvirt dans tout ça ?

La **libvirt** apporte le plan d'administration, de contrôle et de management local de l'hyperviseur. Elle permet de gérer la création, le déplacement, la destruction et plus généralement le cycle de vie des machines virtuelles. Elle n'est pas spécifique à KVM et n'entre pas en jeu dans l'exécution effective de la machine virtuelle. La libvirt reste hors du sujet de cet article, mais il faut globalement la voir comme un plan d'administration, de supervision et de contrôle. Elle fait également le lien avec la gestion des réseaux virtuels (e.g. via **OpenVSwitch**) et est l'une des briques de contrôle, de supervision et d'administration nécessaires aux architectures *Cloud* type **OpenStack**.

2. KVM ET LES EXTENSIONS MATÉRIELLES D'AIDE À LA VIRTUALISATION

2.1 Petit historique du HVM

La virtualisation aidée du matériel (HVM) est arrivée globalement en même temps sur Intel (VT-x) et AMD (initialement appelé AMD Pacifica, aujourd'hui AMD-V). À ses débuts, les mécanismes matériels d'aide à la virtualisation ont entraîné un certain rejet de la part des fournisseurs de solutions de virtualisation, **VMWare** en tête **[3]**, qui la jugeait trop rigide et moins performante que ses propres implémentations logicielles.

Néanmoins, et VMWare l'a tout à fait accepté depuis, les évolutions successives et les nouvelles fonctions d'aide à la virtualisation portées par le matériel ont permis de simplifier les implémentations logicielles des hyperviseurs, au profit d'un accroissement de complexité dans le matériel.

Historiquement les premiers, AMD et Intel ne se sont jamais mis d'accord sur une API commune, ce qui a impliqué pour les fournisseurs d'hyperviseurs (KVM compris) d'implémenter le support des deux API en parallèle.

De nos jours, la richesse et la modularité de l'API matérielle d'aide à la virtualisation génèrent une certaine complexité dans l'implémentation de l'hyperviseur, ce dernier devant vérifier pour chaque fonction d'aide à la virtualisation si celle-ci est supportée par le matériel ou s'il doit intégrer une gestion logicielle de la fonction. C'est le cas des EPT ou des *Shadow Page-Tables* Intel, décrites plus loin.

2.2 Étude des API Intel

2.2.1 L'API VT de virtualisation du cœur processeur

L'API de virtualisation VT-x, permettant la virtualisation de l'exécution du logiciel sur un cœur processeur, a été pour la première fois implémentée en 2005 dans les Pentium 4 modèles 662 et 672. Ce support est visible sous le nom **`vmx`** dans le fichier **`/proc/cpuinfo`** sous GNU/Linux :

Terminal

```
$ cat /proc/cpuinfo |grep vmx | head -1
flags  : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb
rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology
nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx
est tm2 ssse3 fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt
tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch ida
arat epb xsaveopt pln pts dtherm tpr_shadow vnmi flexpriority ept vpid
fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap
clflushopt
```

La principale modification est la création d'un mode supplémentaire dans le « ring » **0** (terminologie spécifique à x86), que l'on trouve classiquement dans la littérature sous le nom de mode **vmx-root**, ou mode hyperviseur.

Pour rappel, une architecture x86 standard possède quatre niveaux d'exécution, allant de **0** à **3** (du plus élevé au plus faible en termes de droits d'accès). Classiquement sous GNU/Linux, le noyau s'exécute en ring **0**, les processus applicatifs en ring **3**. L'apparition du mode **vmx-root** permet de séparer le niveau d'exécution du noyau en deux modes :

- un mode dans lequel les instructions de gestion de la virtualisation sont accessibles : le mode **vmx-root** ;
- un mode dans lequel les instructions de gestion de la virtualisation ne sont pas accessibles et où le matériel considère que le logiciel est cloisonné dans un environnement virtuel distinct du mode **vmx-root** : le mode **vmx non-root**.

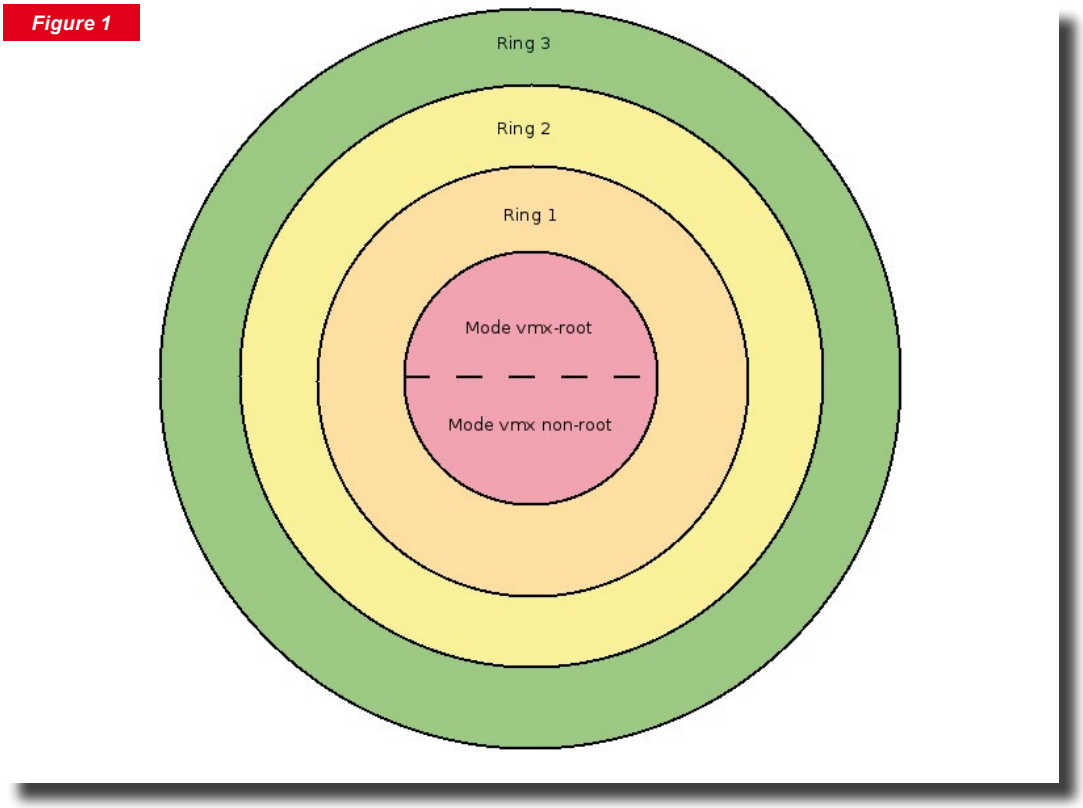
Au démarrage de l'équipement physique, lorsque le mode VMX est activé par le BIOS ou l'UEFI, ce dernier passe dans le mode **vmx-root**, ce qui permet au noyau Linux de pouvoir créer des machines virtuelles. Le support du mode **vmx-root** est cependant désactivable dans le BIOS ou l'UEFI, désactivant tout simplement le support des extensions matérielles de virtualisation des cœurs processeurs, ainsi que la capacité à instancier un mode **vmx non-root** (via l'instruction **VMXON**).

Lorsque le noyau de l'OS invité s'exécute dans un contexte **vmx non-root**, il s'exécute alors avec des droits amoindris, mais néanmoins suffisants pour permettre à un noyau d'un système d'exploitation non modifié de pouvoir s'exécuter sans générer un grand nombre de *traps* liés à son exécution nominale (configuration de la MMU, etc.) - voir figure 1. Ce contexte limite cependant ses droits au contrôle du contexte de la machine virtuelle dans laquelle il s'exécute exclusivement, sans pour autant impacter l'état de la machine physique. Ce dernier point reste la pierre angulaire de toute solution de virtualisation : apporter un mécanisme (logiciel, matériel) permettant de cloisonner les états de la machine physique et de chacune des machines virtuelles de manière performante et sûre.

Le support des contextes d'exécution des machines virtuelles (sous la forme de blocs de mémoire en charge de conserver les contextes des machines virtuelles et quelques informations associées) est fait via des structures nommées VMCS : *Virtual Machine Control Structure*.

Une VMCS est associée à une machine virtuelle. Elle héberge principalement un duplicata d'un sous-ensemble des registres processeur, principalement les registres d'état, permettant de garder en mémoire un certain nombre d'informations sur son contexte lorsque la machine virtuelle est préemptée, pour pouvoir être ensuite rechargée. On y trouve donc typiquement les registres de contrôle, de debug, les registres de pointage des tables de descripteurs GDTR et IDTR, les registres MSR, et ainsi de suite. La VMCS héberge également le contexte du VMM, ceci afin de sauvegarder/recharger rapidement le contexte hyperviseur lorsque la machine virtuelle est préemptée (par exemple, lorsque le logiciel s'exécutant dans le contexte virtuel exécute une instruction nécessitant une intervention de l'hyperviseur).

Figure 1



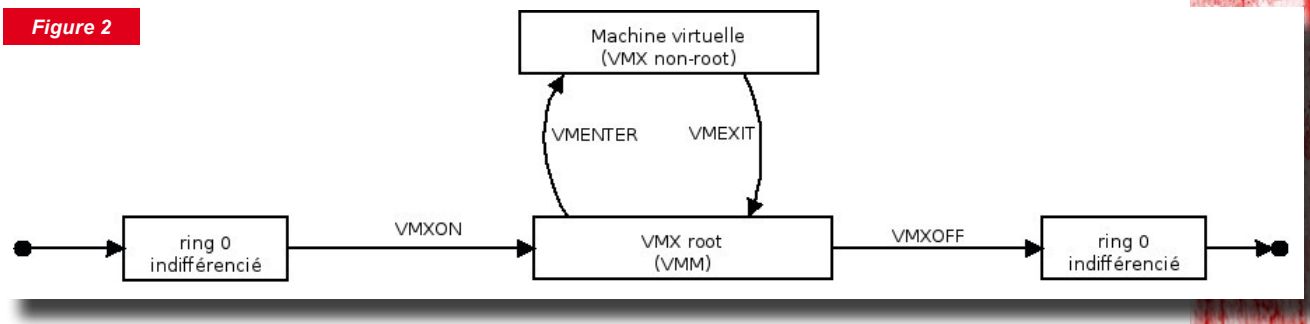
Nouvelle découpe des niveaux d'exécution sur les architectures Intel x86 avec l'apparition de VT-x. Le mode VMX root permet d'héberger le VMM, le mode VMX non-root permet d'héberger le noyau du système d'exploitation de la machine virtuelle sans nécessiter de modification ni provoquer un trop grand nombre de traps liés à son exécution.

L'automate à état (simplifié) lié à l'utilisation de VT-x est présenté en figure 2.

En 2008, Intel ajoute le support de l'aide à la virtualisation de la gestion mémoire et de la MMU : les *Extended Page Tables* (EPT). Ce support est nécessaire pour pouvoir exécuter un processeur virtuel (i.e. un contexte de machine virtuelle, suite à un **VM_ENTER**, confère plus loin) en mode réel (en mode sans MMU, mode dans lequel une machine virtuelle va s'initialiser puis configurer la MMU pour entrer en mode protégé (avec support des pages mémoires et de la segmentation). Le support complet du mode réel dans un contexte processeur virtuel a été intégré en 2010.

Les EPT permettent ainsi à la MMU de différencier la vision des adresses virtuelles et physiques de l'hôte de celles vues par l'OS invité. L'OS invité peut alors configurer la MMU sans pour autant mettre en danger la configuration de la mémoire virtuelle associée à l'espace d'adressage de la machine virtuelle elle-même.

Figure 2



Automate à état simplifié des contextes processeurs sur architecture Intel x86 avec le support des extensions VMX (VT-d).

Globalement, on peut voir l'EPT comme « Inception » : la gestion de tables de pages (celles des processus de l'OS invité) à l'intérieur d'une table de pages (celle de l'espace d'adressage de la machine virtuelle) **[4]**.

2.2.2 L'API VT de cloisonnement des périphériques (I/OMMU)

RAPPEL SUR L'ARCHITECTURE INTEL X86

Historiquement, les architectures Intel possèdent deux *bridges*, en séquence, permettant de relier le processeur aux périphériques, appelés « North-bridge » et « South-bridge ».

Le North-bridge est le plus proche du processeur, il interconnecte les périphériques impliquant un grand nombre d'interactions avec les cœurs (typiquement les contrôleurs mémoire ou le contrôleur graphique via un bus PCI-express dédié).

Le South-Bridge, plus éloigné, interconnecte les autres périphériques (contrôleurs PCI-express complémentaires, contrôleurs USB, etc.).

L'aide à la virtualisation des périphériques physiques, nommée VT-d dans les processeurs Intel le supportant, a pour but d'apporter une fonction de filtrage et de cloisonnement mémoire des périphériques. L'I/OMMU a été positionné au niveau du North-bridge, en aval des contrôleurs mémoire, mais en amont des périphériques à accès rapide (reliés au North-Bridge). Cela permet de filtrer l'accès de tout périphérique qu'il soit relié au North-bridge ou au South-bridge à la mémoire, en implémentant une méthode d'abstraction de la mémoire semblable à de la pagination mémoire, mais dont les destinataires sont les périphériques.

L'I/OMMU est sous le contrôle du processeur en mode `vmx-root`. Cela permet à l'hyperviseur, lorsqu'il alloue un périphérique physique à une machine virtuelle (via un *remapping* des accès au périphérique à cette dernière), de s'assurer que le périphérique ne peut accéder qu'à l'espace mémoire correspondant à la machine virtuelle.

En effet, si l'OS invité envoie un ordre de recopie mémoire au périphérique, il peut lui demander de recopier une zone mémoire entre deux emplacements de la RAM, (typiquement si le périphérique est ou possède un contrôleur DMA). Le périphérique étant alors autonome pour la recopie, la MMU et l'hyperviseur ne peuvent à eux seuls limiter ses accès. Le périphérique est alors capable d'aller lire (voire écrire) dans l'espace mémoire de l'hyperviseur, de l'hôte, ou d'une autre machine virtuelle sans aucun filtrage. L'I/OMMU apporte ce filtrage en permettant à l'hyperviseur de configurer la fenêtre mémoire à laquelle le périphérique a accès.

Cette configuration est assez semblable à la configuration de la MMU pour le processeur : via une abstraction de la mémoire physique via la définition de pages mémoire virtuelles, on ne montre aux périphériques qu'un sous-ensemble de la mémoire ne correspondant qu'aux espaces mémoire physiques auxquels il a l'autorisation d'accéder.

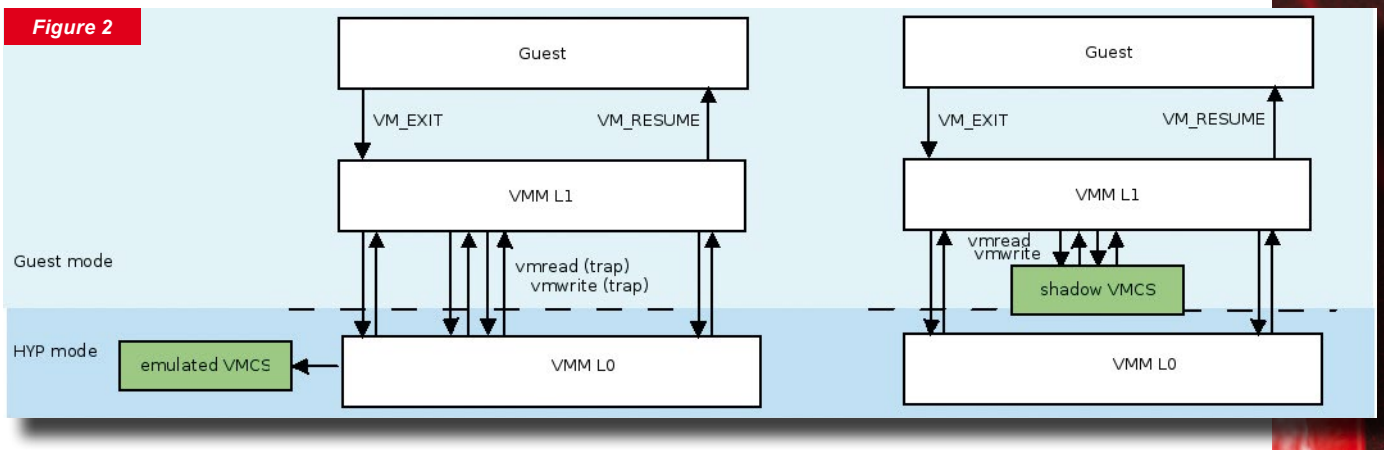
Donner un accès à un périphérique physique à une machine virtuelle est quelque chose d'assez fréquent, classiquement pour des raisons de performances. Cela implique cependant, sans aide de leur part, de les dédier à une machine virtuelle donnée. Cependant, ces dernières années, certains périphériques commencent à supporter nativement la virtualisation, comme c'est le cas des contrôleurs Ethernet compatibles SR-IOV dont nous parlons plus loin dans le cas de KVM.

2.2.3 Petit détour par la « Nested-vitualization »

Le principe de la « Nested Virtualization » est d'exécuter un hyperviseur (*Virtual Machine Monitor*)... par-dessus un autre hyperviseur. On parle alors de VMM L1 (celui du dessus) et de VMM L0 (celui du dessous, qui s'exécute en mode hyperviseur). Ce besoin est apparu plus récemment suite à des besoins plus particuliers d'applications nécessitant une exécution dans des contextes virtualisés, eux-mêmes exécutés en concurrence avec d'autres contextes sur une même plateforme matérielle.

Le problème principal de cette architecture est l'impact sur les performances liées aux *traps* que génère le VMM L1 lorsqu'il traite les contextes des machines virtuelles qu'il gère (`vmenter`, `vmexit`, etc.), impliquant une exécution du VMM L0 qui prendra alors en charge l'exécution effective des traitements hyperviseurs, avant de rendre la main au VMM L1. Dans ses processeurs récents (Haswell), Intel a enrichi son implémentation matérielle de support à la virtualisation pour permettre la gestion des *Shadow VMCS*. Le principe est de gérer des VMCS pour les machines virtuelles de niveau L2 (gérée par le VMM L1) sans impliquer un *trap* vers le VMM de niveau L0. Le gain est de l'ordre de 40 %. L'implémentation dans KVM du support des *Shadows VMCS* a été introduite dans les noyaux 3.10 en avril 2013. Vous pouvez voir en figure 3 une comparaison des interactions entre machines virtuelles avec et sans support des *Shadows VMCS*.

Figure 2



Comparaison des interactions entre machines virtuelles (guest), VMM L1 et VMM L0 avec et sans support des Shadows VMCS. Seuls les ordonnancements de machines virtuelles (VM_EXIT/VM_RESUME) impliquent un passage par le VMM L0 lorsque les Shadows VMCS sont supportées par le matériel.

2.3 Et sur ARM alors ?

L'ÉCOSYSTÈME ARM

ARM est une société anglaise ayant connu une forte croissance ces dernières années avec le marché des smartphones et des objets connectés. ARM ne crée pas de processeur, mais des *designs*, que d'autres (Samsung, AllWinner, etc.) se chargent d'intégrer à des SoC (*System on Chip*), en y ajoutant divers composants matériels supplémentaires.

La famille Cortex-A de ARM est la plus puissante, apte à exécuter des OS riches type GNU/Linux. Les dernières versions du jeu d'instruction sont l'ARMv7 (32 bits) et ARMv8 (64 bits). Les cœurs ARM aptes à gérer la virtualisation sont les Cortex A7 (ARMv7a), A15 (ARMv7a), A53 (ARMv8), A57 (ARMv8) et A72 (ARMv8).

Globalement, vous avez probablement plus de composants ARM chez vous que d'x86, parfois sans le savoir (passerelles multimédias, smartphones, tablettes, téléviseurs, montres connectées, NAS, etc.).

L'arrivée du support de la virtualisation matérielle sur ARM est plus récente (ARMv7a, présenté par ARM en 2010). La virtualisation du cœur processeur, correspondant à l'apparition d'un nouveau mode d'exécution, appelé mode hyperviseur (**`HYP mode`** dans la littérature) a nécessité un certain travail de la part de l'équipe KVM :

- une plus forte dissociation entre le module KVM et le noyau Linux ; seul le module KVM devant s'exécuter en mode hyperviseur, le noyau Linux restant en mode superviseur (à l'inverse du mode **`vmx-root`** Intel) ;
- une modification de la chaîne de boot du noyau. Inutile sous x86, le noyau, sur les architectures ARM, doit démarrer en mode hyperviseur, charger le module KVM, puis passer en mode superviseur.

La première version de KVM avec le support de la virtualisation ARM (sur base ARMv7a) est sortie en 2014 **[5]**.

Sur ARM, la séparation noyau/KVM est plus flagrante **[6]**, bien que les deux continuent bien sûr d'interagir pour l'ordonnancement des machines virtuelles (le module KVM n'ordonnance pas à proprement parler) ou pour la gestion des I/O.

Les spécificités de l'implémentation de la virtualisation du cœur processeur sur ARM ont nécessité de *patcher* également le bootloader (classiquement, U-boot) pour que ce dernier démarre l'OS en mode hyperviseur et non plus en mode superviseur. En effet, il n'y a aucun retour arrière possible. Le patch d'U-boot date initialement de 2013, pour le support des extensions de virtualisation ARMv7a (Cortex A15, puis A7, comme la CubieBoard 2). Le patch a été complété pour l'ARMv8 par la suite.

Avec l'arrivée des Cortex A53/A57 et A72, ARM a déployé un mécanisme appelé SMMU (*System-MMU*) dont le but est d'apporter un cloisonnement des périphériques comme le fait l'I/OMMU sous Intel. La différence majeure est dans sa philosophie :

- Intel positionne l'I/OMMU en coupure entre la mémoire et les périphériques ;
- ARM (tout comme certains PowerPC) positionne la SMMU de manière distribuée : chaque contrôleur pouvant être initiateur sur l'*Interconnect* (apte à communiquer directement avec d'autres périphériques sans passage par la MMU et sans ordre initié par le processeur) est cloisonné derrière une instance de SMMU, qui sera en charge de faire de la translation d'adresse et de la gestion de droits. Sous certains PowerPC (typiquement chez NXP), le PAMU (*Peripherical Access Management Unit*) a le même but.

ATTENTION !

Nous parlons ici des extensions de virtualisations ARM. Il ne s'agit pas de la capacité *TrustZone*, beaucoup plus ancienne et dont le but n'est pas le même.

2.4 Les autres architectures

KVM supporte également l'architecture PowerPC, du moins pour les SoC qui supportent les extensions de virtualisation matérielle (par exemple, certains SoC de NXP). La communauté open source y est cependant moins active, car ce marché est plus confidentiel, limité principalement à certains secteurs de l'industrie comme les télécoms.

2.5 Ok, mais plus concrètement… comment KVM gère des machines virtuelles ?

Nous avons vu la manière dont KVM interagit avec le matériel pour construire les éléments structurels en charge de différencier les contextes virtuels des contextes de l'hôte. Néanmoins, il y a encore du travail logiciel à réaliser.

2.5.1 La création d'une machine virtuelle

Une machine virtuelle est vue comme un processus de l'hôte… ou presque. Elle possède un espace d'adressage (au sens de la MMU), mais avec une structure particulière. Là où un processus de l'hôte est classiquement découpé en deux types de *mapping* mémoire (*user* et *kernel*, pour la gestion des *traps* et des appels systèmes), l'espace d'adressage d'une VM possède un *mapping* mémoire taggué… « guest ». L'OS hôte n'est alors pas en charge d'en gérer le contenu **[7]**. Lorsque nous parlons de l'espace d'adressage de la machine virtuelle, il faut bien voir qu'il ne s'agit pas de l'espace d'adressage du processus Qemu qui lui est apparenté, mais bien de la machine virtuelle elle-même. Il s'agit bien de deux espaces d'adressages différents, même si un certain *remapping* mémoire est fait entre les deux. En effet, le processus Qemu est lui un processus normal de l'OS hôte, et doit donc avoir un *mapping user* et *kernel*, comme tous les autres.

La création effective de la machine virtuelle se fait au travers du fichier spécial **`/dev/kvm`**, au travers duquel des ordres liés à la création/suppression de machines virtuelles peuvent être envoyés. Ces ordres sont envoyés classiquement par qemu (et non directement par la libvirt), lorsque ce dernier est configuré pour travailler avec KVM.
La séquence est la suivante :

- Qemu fait alors une demande explicite de création de machine virtuelle au module KVM ;
- ce dernier initialise les éléments matériels (structure VMCS, EPT si présent, etc.) ;
- le module KVM crée également les contextes logiciels (structures de données du noyau, etc.) et interagit avec les éléments du *kernel* comme l'ordonnanceur ou le gestionnaire mémoire pour créer l'espace d'adressage de la machine virtuelle et l'instancier ;
- la VM démarre suite à un appel à l'instruction **`VMENTER`**. La séquence de *boot* est gérée par Qemu, qui fournit par ailleurs une image BIOS (cas x86 uniquement). Globalement, Qemu sera en charge de traiter toute I/O ou signal lié au matériel émulé. Cependant, ces traitements ne se font pas par un passage direct entre la machine virtuelle et Qemu : toute sortie de la machine virtuelle (**`VMEXIT`**) implique un passage par le VMM (le module KVM), qui est alors en charge de déterminer si la sortie est liée à une action à effectuer de la part de Qemu ou non (il peut s'agir d'un *trap* impliquant une action du module KVM) - voir figure 4 (page suivante).

2.5.2 La destruction d'une machine virtuelle

La destruction est assez logique, car elle correspond à la suppression du processus Qemu en charge de gérer les I/O émulées de la machine virtuelle ainsi que du contexte logiciel et matériel associé à la machine virtuelle dans le noyau Linux et dans le matériel.

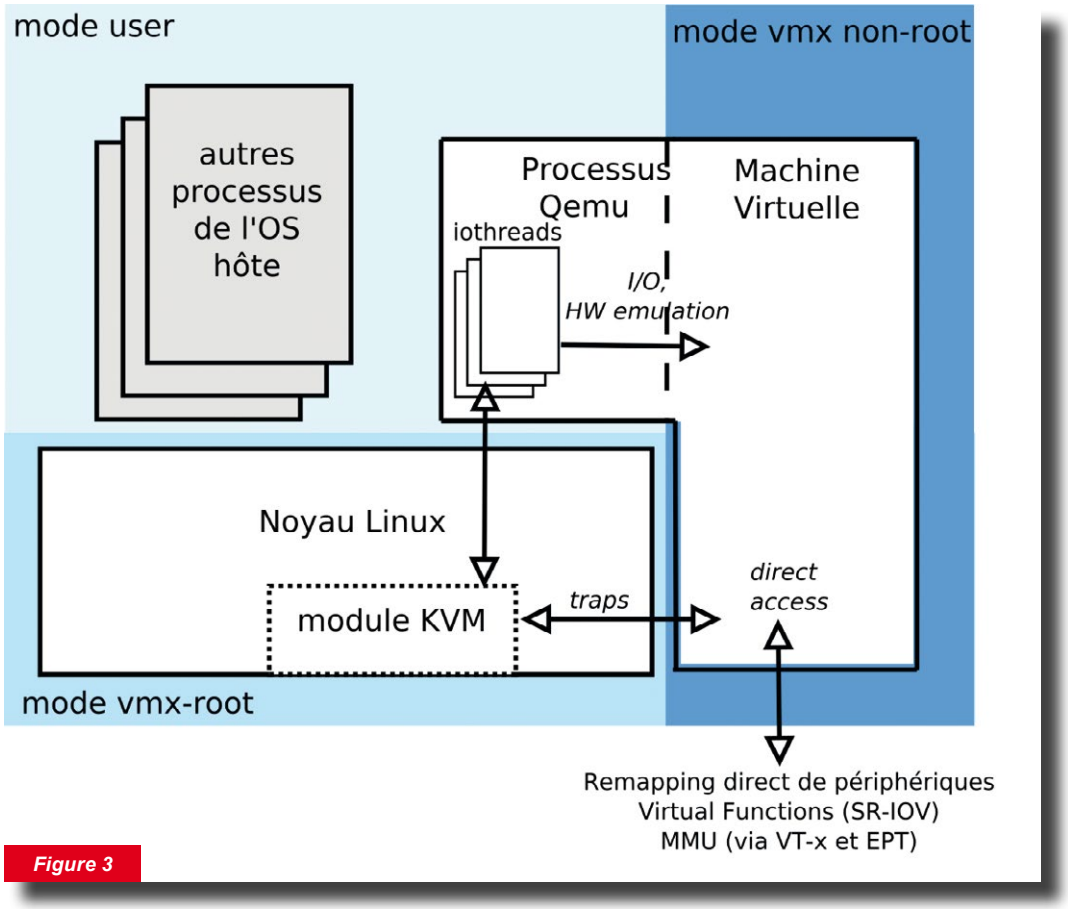


Figure 3

Architecture logicielle générale d'un hôte KVM hébergeant une machine virtuelle et interactions.

Cependant, on utilise classiquement les capacités de Qemu à envoyer un message ACPI (ou équivalent) à l'invité avant sa destruction afin de lui permettre de « s'éteindre » proprement. C'est la différence entre le « shutdown » et le « destroy » dans la configuration XML de la libvirt. Cet ordre n'est pas géré par KVM, mais par la libvirt par interaction avec Qemu. KVM se charge lui de supprimer la machine virtuelle, pas de gérer le logiciel qui s'y trouve cloisonné.

2.5.3 L'ordonnancement d'une machine virtuelle

Le module KVM utilise les capacités du noyau Linux pour ordonnancer les machines virtuelles. Cela lui permet de gérer l'affinité CPU, les cgroups et ainsi de suite. Il est alors possible d'exploiter des capacités évoluées d'ordonnancement, paramétrable par exemple dans la configuration d'un domaine libvirt, pour spécifier le quantum CPU autorisé pour chaque VM, l'affinité ou divers autres éléments d'ordonnancement et d'optimisation.

2.5.4 Mais alors… Les processus root de l'hôte voient et ont accès aux machines virtuelles ?

En réalité la réponse est non. Ils voient et ont potentiellement accès au processus Qemu, mais pas à l'espace d'adressage de la machine virtuelle (celui taggué « guest »). Celui-là, seul le noyau Linux est capable de le voir.

3. LE MODÈLE VIRTIO

L'API Virtio est une implémentation dont le but est de fournir une abstraction de plusieurs familles de périphériques au travers d'une API unifiée, afin d'optimiser les performances d'accès par rapport à l'émulation d'un périphérique. Il ne s'agit plus ici d'émuler un composant, mais réellement d'employer volontairement une API logicielle pour instancier des périphériques divers comme de type bloc ou de type réseau (**virtio-block** et **virtio-net**). L'usage de l'API Virtio implique bien sûr de modifier le noyau de l'OS invité. Il doit en effet volontairement utiliser un périphérique de type Virtio. Il s'agit donc ici de paravirtualisation des I/O. Attention cependant : KVM, bien qu'apte à supporter la paravirtualisation des I/O, ne sait pas paravirtualiser le cœur processeur comme le permet par exemple Xen : la présence des extensions matérielles d'aide à la virtualisation du cœur processeur sont nécessaires.

L'usage de composant virtio permet de créer des périphériques virtuels en considérant que tout mécanisme de communication avec un périphérique d'I/O revient au final à un échange de *buffers*. On retrouve donc un ensemble limité de fonctions permettant de gérer la transmission et la réception de *buffers*, ainsi que des fonctions de gestion d'événements. Son implémentation est donc basée sur une abstraction :

- les drivers virtio spécialisés (driver Ethernet, de disque, etc.) ;
- un backend « générique » de traitement des *buffers*.

Côté Qemu, les mécanismes de traitements des buffers doivent être également traités, ce qui implique donc également la présence de code spécifique pour récupérer dans l'*iothread* (voir figure 4) les *buffers* transmis par la machine virtuelle ou devant lui être remis. À cela, il faut également déterminer vers quelle destination doivent être émises ces données. Il s'agit là des *backend-drivers*. Il en existe un grand nombre :

- à destination d'un fichier pour un périphérique en mode bloc géré sous forme d'un fichier ;
- à destination de la pile réseau du noyau Linux (cas d'un périphérique virtio-net interconnecté à un *bridge* par exemple) ;
- à destination d'un autre processus *userspace* (typiquement le cas de DPDK **[8]**) ;
- ...

Il faut bien voir que ces mécanismes impliquent l'exécution du processus qemu et de son ou ses *iothreads*, et donc classiquement une préemption de la machine virtuelle. À l'inverse, le *remapping* direct de périphériques physiques ou de Virtual Functions SR-IOV évite ces préemptions.

4. KVM ET LES PÉRIPHÉRIQUES SR-IOV

SR-IOV (*Single-Root I/O Virtualization*) définit un ensemble de méthodes pour les périphériques PCIe afin de les rendre compatibles avec les architectures virtualisées. Il s'agit d'une implémentation matérielle et donc d'un support que doit posséder le périphérique.

Le périphérique est alors apte à instancier des contextes « virtuels » de lui-même (appelés VF, *Virtual Functions*), en plus d'un contexte physique (PF, *Physical Function*) sous contrôle de l'hôte. Un périphérique peut alors être utilisé par plusieurs machines virtuelles en appliquant un *remapping* des VF dans les machines virtuelles tout en assurant un cloisonnement entre ses dernières.

Bien que la spécification SR-IOV décrit qu'un périphérique est apte à instancier jusqu'à 128 VF, dans les faits on retrouve plutôt de l'ordre de 8 VF par PF. Par exemple, les contrôleurs Ethernet Intel modernes comme les cartes i240 (driver **igb**) supportent cette fonction.

La configuration des VF doit être faite par l'hôte et n'est pas du ressort du module KVM. Une fois configurée, chaque VF est vue comme un net *device* (au sens Linux), ce qui permet par la suite de l'assigner de manière tout à fait naturelle aux machines virtuelles. Chaque VF possède son propre contexte et ses registres de configuration, paramétrables par la machine virtuelle, avec des limitations pour ne pas impacter l'état du contrôleur physique (typiquement une modification des paramètres d'auto-négociation pouvant entrer en collision avec une autre VF, celle-ci étant reliée au même contrôleur Phy et au même câble Ethernet).

5. KVM ET LA SÉCURITÉ

La présence de plusieurs machines virtuelles sur une même machine physique implique un grand nombre d'impacts en terme de sécurité dont on ne parlera pas ici. Cependant, il faut bien voir qu'il est important :

1. de cloisonner les machines virtuelles entre elles ;
2. de cloisonner l'ensemble des fonctions de virtualisation (libvirt, qemu, etc., selon la richesse des fonctions présentes) du reste des fonctions logicielles de l'hôte ;
3. de protéger le noyau Linux, qui héberge entre autres le module KVM.

Les machines virtuelles, pour interagir avec l'environnement logiciel qui les porte, s'appuient sur le processus `qemu`. Ce dernier doit donc impérativement être restreint pour limiter le risque de VM-Escape (capacité, pour une fonction logicielle virtualisée, de sortir de son contexte virtualisé). La libvirt apporte ainsi des éléments d'aide à la sécurisation en simplifiant le cloisonnement des machines virtuelles via un driver SELinux et via les cgroups. Le but est de limiter les accès du processus Qemu, en charge de gérer les I/O de la machine virtuelle.

Lorsque des périphériques physiques sont *remappés* dans une machine virtuelle, la présence d'une I/OMMU (ou d'une SMMU pour ARM, PAMU pour PowerPC) est impérative. Cette fonction doit être configurée de manière rigoureuse.

Classiquement, l'OS hôte est dédié à la virtualisation. Il peut néanmoins héberger des services divers avec des droits étendus, et il est important de s'assurer que les fonctions n'ayant pas nécessité d'accéder aux fonctions de gestion de la virtualisation restent contraintes. Les LSM (**AppArmor**, **SELinux**, etc.) permettent, comme pour tout système GNU/Linux de mettre en place de telles entraves. La présence d'un OS hôte en plus d'une fonction d'hyperviseur accroît le risque sécuritaire.

Des entités comme l'ANSSI ont défini des documents de bonnes pratiques pour les serveurs et les systèmes de virtualisation, qu'il ne faut pas hésiter à étudier [9][10].

CONCLUSION

Nous avons pu voir dans cet article que l'hyperviseur KVM est basé sur une architecture faisant intervenir de multiples entités logicielles comme matérielles, ce qui rend sa compréhension et sa maîtrise difficile. De plus, l'hétérogénéité et la modularité des architectures matérielles rendent l'implémentation d'un hyperviseur performant et portable malaisé malgré l'aide que le matériel peut apporter.

Cet article a mis l'accent sur les différentes API et différents chemins de données cachés aux utilisateurs de la technologie KVM, afin de mieux appréhender la complexité sous-jacente. Il ne faut cependant pas perdre de vue que les technologies logicielles décrites dans cet article ne sont qu'un composant de l'écosystème logiciel qui fait aujourd'hui la force de KVM et qui intègre des fonctions de plus haut niveau comme **OpenStack**, permettant d'aboutir à des infrastructures de virtualisation de grande envergure et répondant à des contraintes métier que l'hyperviseur seul ne permet pas de résoudre. ■

RÉFÉRENCES

[1] KVM Myths - Uncovering the Truth about the Open Source Hypervisor : **https://www.ibm.com/developerworks/community/blogs/ibmvirtualization/entry/kvm_myths_uncovering_the_truth_about_the_open_source_hypervisor?lang=en**

[2] Red Hat, *« Top-secret KVM, lessons learned from an ICD 503 deployment »* : **http://people.red-hat.com/jamisonm/usaf/KVM_security.pdf**

[3] VMWare, *« Understanding Fullvirtualization, paravirtualization and hardware assist »* : **https://www.vmware.com/files/pdf/VMware_paravirtualization.pdf**

[4] *« EPT in KVM »* : **https://zhongshugu.wordpress.com/2010/06/17/ept-in-kvm/**

[5] *kvm : the Linux Virtual Machine Monitor*

[6] *KVM/ARM : The Design and Implementation of the Linux ARM Hypervisor*

[7] Intel, *« Intel® 64 and IA-32 Architectures Software Developer's Manual »*, Volume 3B - *« System Programming Guide, Part 2 »*.

[8] DPDK, kit de développement pour application de traitement réseau haute performance : **http://www.dpdk.org/**

[9] *« Bonnes pratiques de configuration d'un poste GNU/Linux »* : **http://www.ssi.gouv.fr/entreprise/guide/recommandations-de-securite-relatives-a-un-systeme-gnulinux/**

[10] *« Bonnes pratiques de sécurisation d'un système de virtualisation »* : **http://www.ssi.gouv.fr/entreprise/guide/problematiques-de-securite-associees-a-la-virtualisation-des-systemes-dinformation/**

4

4 SÉCURISEZ

À découvrir dans cette partie...

4.1 La sécurité du noyau avec SELinux

Dans cet article, nous allons aborder un aperçu de l'architecture LSM intégrée au noyau Linux permettant en conséquence de déployer SELinux. Ensuite, nous détaillons comment activer et surveiller la sécurité des services grâce à SELinux et nous prenons comme exemple le serveur web Apache tout en parcourant toutes les étapes d'administration pour la mise en place d'une politique renforcée protégeant le système des accès non autorisés. Nous clôturons l'article par la résolution de quelques problèmes typiques liés à SELinux que vous pouvez rencontrer tout au long de la phase de déploiement de la sécurité SELinux.p. 104

LA SÉCURITÉ DU NOYAU AVEC SELINUX

Issam MEJRI

Le noyau Linux dispose d'une couche de sécurité lui permettant de se défendre des éventuelles attaques et de protéger ainsi les données des utilisateurs contre les systèmes dont la sécurité a été compromise. Cette couche de sécurité se manifeste sous la forme d'un Framework appelé LSM (Linux Security Module). Ce dernier permet de supporter différents modules de sécurité et parmi ces modules on peut citer SELinux (Security Enhanced Linux). SELinux renforce la sécurité système par le biais de règles de sécurité simples à mettre en œuvre, donnant le choix à l'administrateur de déployer la politique de sécurité qui lui convient le mieux et qui dépendra de ses besoins et de la stratégie à suivre.

SELinux est développé dans les locaux de la **NSA** (*National Security Agency*). Ayant le besoin de renforcer la sécurité des données et des informations top secrètes, le service de renseignement américain (nommé NSA) était dans la nécessité de développer un produit lui permettant d'assurer un haut niveau de sécurité pour les données sous forme de **MLS** (*Multi-Level Security*), cette notion de sécurité consistant à classifier les objets suivant leur catégorie et leur confidentialité. C'est la raison pour laquelle SELinux a vu le jour. Une autre approche de sécurité fondamentale sur laquelle repose SELinux est le mécanisme de sécurité de type **MAC** (*Mandatory Access Control*) qui sera détaillé dans la suite de l'article.

La distribution de SELinux par la NSA sous la licence GPL, la modularité et la robustesse de **LSM** (*Linux Security Module*) dans le noyau Linux, font de lui un outil incontournable pour la sécurité d'un système **GNU/Linux** hébergeant des données sensibles.

SELinux est présent nativement dans la plupart des distributions de la famille **Red Hat** comme **Fedora**, **RHEL** et **CentOS**. Le noyau des distributions **Debian** est compilé avec la prise en charge de SELinux, mais il est désactivé par défaut. J'aborderai à la fin de l'article comment activer SELinux dans Debian et ainsi satisfaire tous les goûts !

1. LE CONTRÔLE D'ACCÈS SOUS LINUX

Avant d'entrer dans le vif du sujet, il est judicieux de comprendre les différents types de contrôle d'accès implémentés dans les systèmes Linux, pour savoir quel avantage apportera SELinux par rapport aux règles de sécurité classiques Linux/Unix connues en tant que contrôle d'accès discrétionnaire **DAC** (*Discretional Access Control*) alors que ce dernier est basé sur le contrôle d'accès obligatoire MAC.

1.1 La sécurité de type DAC

C'est le modèle de sécurité standard des permissions *user/group/other*. Dans ce mode, un programme est exécuté avec les permissions de l'utilisateur qui l'a lancé. Par exemple, si je me connecte en tant qu'utilisateur `issam` et que j'exécute le programme `mail` pour lire mes messages, le programme s'exécute avec mon *ID* utilisateur et il est désormais capable d'agir sur le système avec l'ensemble des opérations pour lesquelles j'ai une autorisation. Il a la possibilité de lecture/écriture dans mon répertoire personnel ainsi que ses sous-répertoires. Bien évidemment, `mail` ne pourra pas accéder à n'importe quel fichier, mais en exploitant une éventuelle vulnérabilité dans `mail`, un malveillant pourra agir pour que `mail` puisse accéder ou modifier des fichiers sensibles, voire mettre le système lui-même hors service, ou encore compromettre la sécurité de mon compte utilisateur.

On peut conclure qu'avec la sécurité de type DAC, compromettre un programme (processus, *thread*) met en péril l'ensemble des objets dont l'utilisateur possède l'accès. Plus dangereux encore, si un processus est exécuté en tant que root, sachant que root a des droits illimités sur le système, si un intrus prend la main sur ce processus, il est fort probable que votre système et les services fournis seront inexploitables !!!

Bref, DAC est insuffisant pour protéger efficacement un système. Le recours à la sécurité de type MAC s'avère indispensable.

1.2 La sécurité de type MAC

Dans ce mode, la protection d'accès aux objets (fichiers, *sockets*, etc.) est imposée par le système et non pas par le propriétaire des objets, donc la notion de privilèges utilisateurs ne sera pas prise en compte. Concrètement, quand un sujet (processus, *thread*) sollicite l'accès à un objet tel qu'un répertoire par

exemple, l'accès est défini par une règle de sécurité complexe autorisant ou non la demande en question empêchant ainsi les personnes mal intentionnées de s'introduire dans le système par le biais d'une faille de sécurité. SELinux est dédié à ce type d'opération.

SELinux dispose également des contrôles d'accès de type **TE** (*Type Enforcement*), **RBAC** (*Role-Based Access Control*) et **MLS** (*Multi-Level Security*) qui sort de l'objet de cet article.

2. L'ARCHITECTURE LSM (LINUX SECURITY MODULE)

LSM est disponible à partir de la version 2.6 du noyau Linux depuis l'année 2003. Ce *framework* ajoute des *hooks* ou ancres au sein du noyau Linux dans différents endroits, y compris les points d'entrées des appels systèmes, et permet des implémentations de sécurité (à l'instar de SELinux) pour fournir des fonctions spécifiques qui seront interrogées lorsqu'un éventuel *hook* sera sollicité. Ces fonctions pourront alors, en se basant sur une politique de sécurité, en plus d'autres informations, autoriser ou non un tel appel système, ou l'exécution d'une telle action (voir figure 1).

L'aspect universel de LSM offre la possibilité d'héberger plusieurs solutions de sécurité de type MAC telles qu'**Apparmor** qui lui est présent dans les distributions **SUSE Entreprise Linux** et **Ubuntu**. L'indépendance de LSM à une architecture de sécurité particulière a participé pleinement à son développement et à son intégration au noyau Linux.

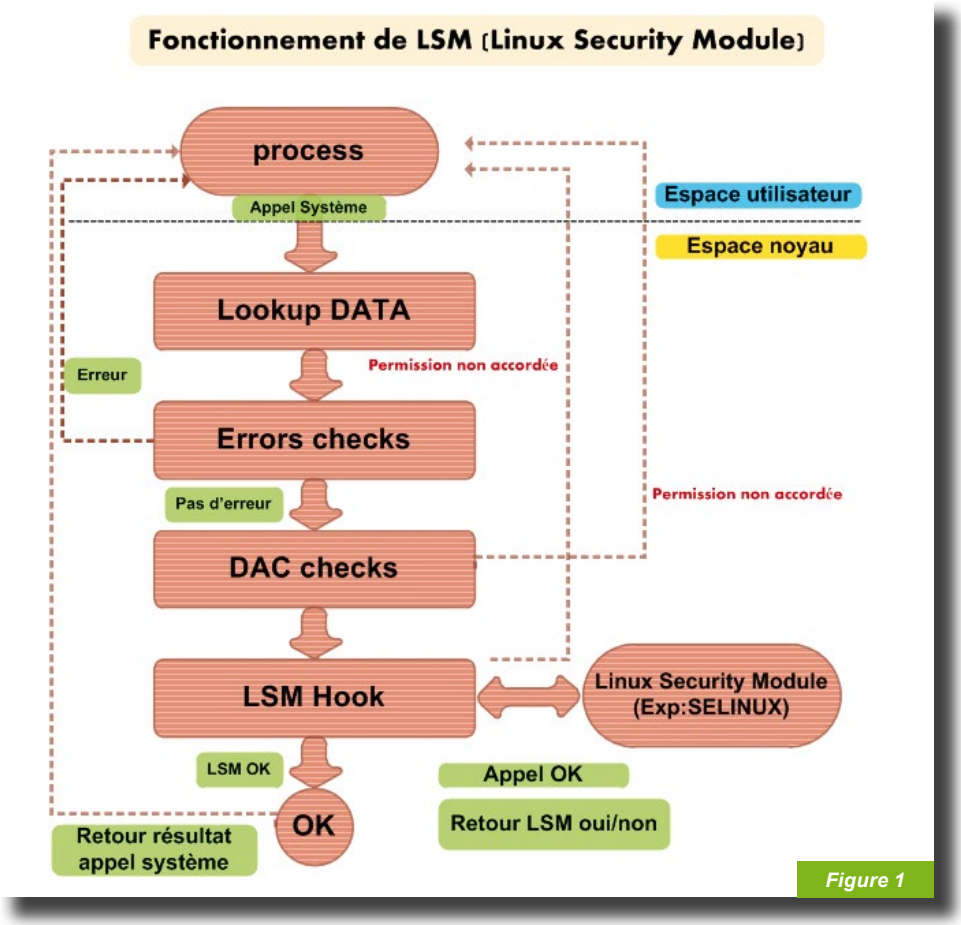


Figure 1

3. CONFIGURATION DU NOYAU

La distribution GNU/Linux utilisée dans cet article est CentOS 7. SELinux est activé par défaut. Cependant, on peut vérifier quelles sont les options de compilation qui ont été prises en charge en inspectant la configuration de la compilation du noyau en se déplaçant dans le répertoire `/usr/src/kernels/Kernel-Version/` et en lançant la commande :

Terminal

```
# make menuconfig
```

La fenêtre de configuration du noyau s'affiche (voir figure 2), sélectionnez ***Security Options*** ; une deuxième fenêtre s'affiche (voir figure 3), descendez jusqu'au ***NSA SELinux Support*** pour voir les différentes options de compilation de SELinux.

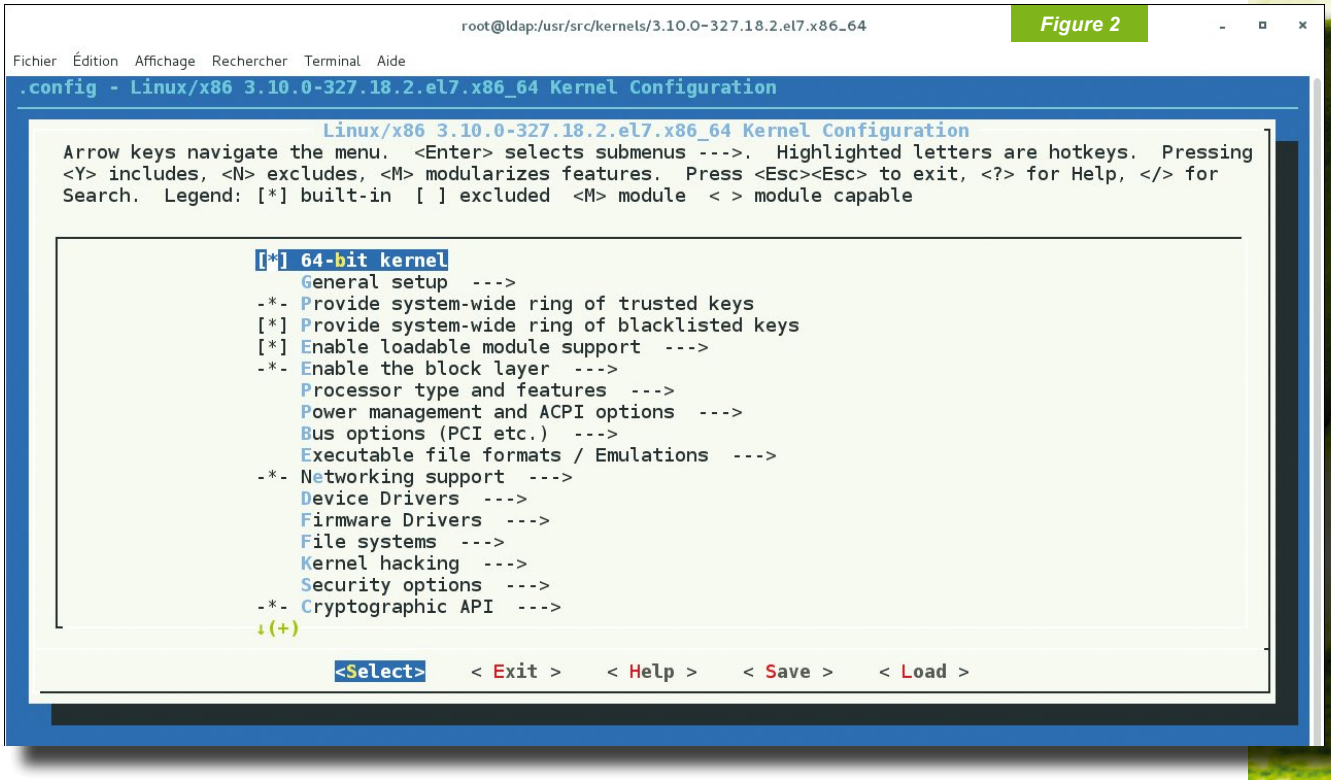


Figure 2

Il est aussi possible d'inspecter le fichier `.config` directement :

Terminal

```
# cd /usr/src/kernels/Kernel-Version/
# cat .config | grep SELINUX
CONFIG_SECURITY_SELINUX=y
CONFIG_SECURITY_SELINUX_BOOTPARAM=y
CONFIG_SECURITY_SELINUX_BOOTPARAM_VALUE=1
CONFIG_SECURITY_SELINUX_DISABLE=y
CONFIG_SECURITY_SELINUX_DEVELOP=y
CONFIG_SECURITY_SELINUX_AVC_STATS=y
CONFIG_SECURITY_SELINUX_CHECKREQPROT_VALUE=1
# CONFIG_SECURITY_SELINUX_POLICYDB_VERSION_MAX is not set
CONFIG_DEFAULT_SECURITY_SELINUX=y
```

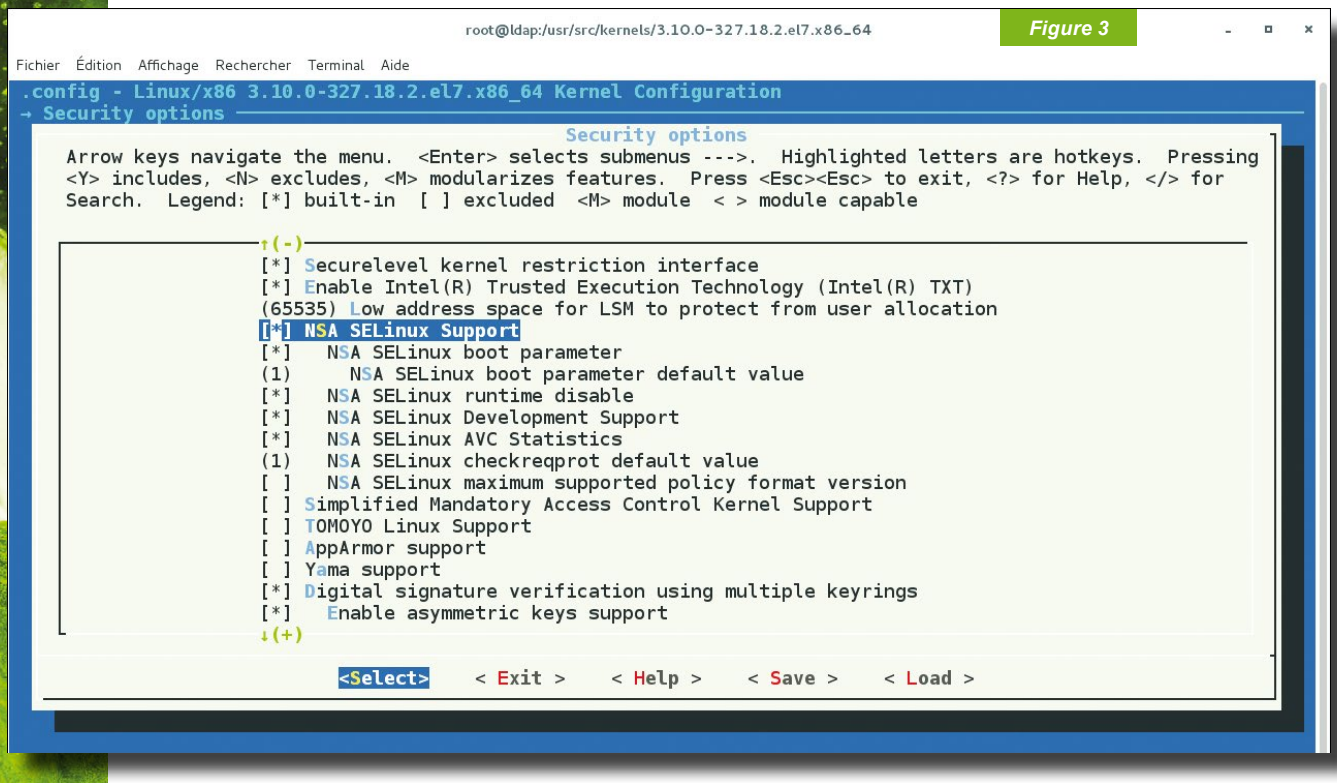


Figure 3

4. PRINCIPE DE FONCTIONNEMENT DE SELINUX

SELinux est un ensemble de règles de sécurité qui détermine quels processus sont autorisés à accéder à quels fichiers, répertoires et ports. Chaque fichier, processus, répertoire et port dispose d'une étiquette de sécurité spéciale appelée **contexte de sécurité SELinux**. Le contexte est un nom qui est utilisé par la politique SELinux pour déterminer si un processus peut ou non accéder à un fichier, à un répertoire ou à un port. Par défaut, la stratégie n'autorise aucune interaction, sauf si une règle explicite accorde un accès. S'il n'existe aucune règle d'autorisation, aucun accès n'est autorisé.

Les étiquettes, appelées encore labels, comportent plusieurs champs : utilisateur, rôle, type et niveau de sensibilité.

Dans CentOS 7 et RHEL 7, la stratégie activée par défaut est la stratégie ciblée (*targeted*). Cette stratégie repose sur le troisième champ qui est le contexte de type pour les fichiers ou domaines pour les processus. Les noms de contexte de type se terminent généralement par `_t`. Une politique de sécurité définit les interactions permises entre domaines et types.

Suivant le contexte de sécurité affecté à un objet, SELinux prend une décision pour un processus qui souhaiterait accéder à cet objet. Dans SELinux, chaque processus doit s'occuper de ses affaires ! Autrement dit, le processus `httpd` (serveur web Apache) par exemple n'a pas le droit de toucher à un répertoire dédié à un partage

FTP et vice-versa (sauf exception explicite à l'initiative de l'administrateur qui autorise ce type d'accès par une règle dédiée ajoutée à la politique par défaut).
Le processus **httpd** peut lire et écrire dans le répertoire **/var/www/html**.
C'est SELinux qui décide si l'accès est autorisé ou non (voir figure 4).

Figure 4

4.1 Contexte de sécurité

Reprenons l'exemple du serveur Apache (processus **httpd**). Le répertoire auquel le processus **httpd** est autorisé à accéder est **/var/www/html**. Quelle étiquette et donc contexte de sécurité possède ce répertoire ? La commande **ls** avec l'option **-Z** permet d'inspecter le contexte de sécurité pour les fichiers et répertoires :

Terminal

```
# ls -lZd /var/www/html
drwxr-xr-x. root root system_u:object_r:httpd_sys_content_t:s0 /var/www/html/
```

On peut conclure que le contexte de sécurité de ce répertoire est **httpd_sys_content**. Les autres champs sont :

⇨ **system_u** : c'est l'utilisateur SELinux ;

⇨ **object_r** : c'est le rôle SELinux.

Démarrons maintenant le serveur Apache (**httpd**) et affichons le contexte de sécurité du processus :

Terminal

```
# systemctl start httpd
# ps axz | grep httpd
system_u:system_r:httpd_t:s0   31191 ?   Ss  0:00  /usr/sbin/httpd -DFOREGROUND
system_u:system_r:httpd_t:s0   31193 ?   S   0:00  /usr/sbin/httpd -DFOREGROUND
system_u:system_r:httpd_t:s0   31194 ?   S   0:00  /usr/sbin/httpd -DFOREGROUND
system_u:system_r:httpd_t:s0   31195 ?   S   0:00  /usr/sbin/httpd -DFOREGROUND
system_u:system_r:httpd_t:s0   31196 ?   S   0:00  /usr/sbin/httpd -DFOREGROUND
system_u:system_r:httpd_t:s0   31197 ?   S   0:00  /usr/sbin/httpd -DFOREGROUND
```

On peut conclure que le contexte de sécurité du processus **/usr/sbin/httpd** (cinq processus clonés en mémoire) est **httpd_t**.

Alors comment ceci est géré par la politique SELinux ? La réponse est simple : par une règle *Allow* autorisant tout sujet de type **httpd_t** d'accéder aux objets de type **httpd_sys_content**.

Pour afficher les règles *Allow* concernant le type **httpd_t**, il faut passer par la commande **sesearch**.

NOTE

La commande sesearch nécessite l'installation du paquet settools-console.

Terminal

```
# sesearch -A|grep httpd_t|grep httpd_sys_content
   allow httpd_t httpd_sys_content_t : file { ioctl read getattr lock
open } ;
   allow httpd_t httpd_sys_content_t : dir { ioctl read getattr lock
search open } ;
   allow httpd_t httpd_sys_content_t : lnk_file { read getattr } ;
   allow httpd_t httpd_sys_content_t : dir { ioctl read write getattr
lock add_name remove_name              search open } ;
```

C'est le résultat attendu puisque **httpd** pourra accéder et réaliser des tâches comme la lecture, l'écriture et l'ouverture des fichiers dont le label est **httpd_sys_content_t**.

Si SELinux n'est pas activé, le serveur web peut parcourir n'importe quelle partie du système de fichiers tel qu'un répertoire personnel d'un utilisateur ou même la racine du système de fichiers. La stratégie ne dispose d'aucune règle au préalable permettant l'accès aux dossiers personnels des utilisateurs (sauf explicitement ou via les booléens, voir plus loin dans l'article) de sorte que l'accès n'est pas autorisé.

4.2 Modes SELinux

Les modes SELinux permettent d'activer ou de désactiver la protection SELinux. Pour activer un mode, il faut passer par un réglage dans le fichier de configuration de SELinux :

Terminal

```
# cat /etc/selinux/config

# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#     enforcing - SELinux security policy is enforced.
#     permissive - SELinux prints warnings instead of enforcing.
#     disabled - No SELinux policy is loaded.
SELINUX=enforcing
# SELINUXTYPE= can take one of three two values:
```

```
#     targeted - Targeted processes are protected,
#     minimum - Modification of targeted policy. Only selected processes
are protected.
#     mls - Multi Level Security protection.
SELINUXTYPE=targeted
```

Il existe trois modes de fonctionnement de SELinux :

- Le mode *enforcing* ou mode *strict* : dans ce mode, les règles de stratégie sont appliquées et les infractions sont consignées dans le journal ;
- Le mode *permissif* : ce mode est souvent utilisé à des fins de dépannage. En mode permissif, SELinux autorise toutes les interactions, même en l'absence de règles explicites, et journalise toutes les opérations qu'il aurait refusées en mode strict. On peut utiliser ce mode pour autoriser temporairement l'accès au contenu dont l'accès est protégé par SELinux. On peut passer du mode strict vers le mode permissif à la volée ;
- Le mode *désactivé* : ce mode désactive complètement SELinux. Il faut redémarrer le système pour désactiver définitivement SELinux ou pour basculer du mode désactivé vers les modes strict ou permissif. Sachez qu'il est déconseillé de désactiver complètement SELinux.

La commande **`getenforce`** affiche le mode SELinux activé :

Terminal

```
# getenforce
Enforcing
```

Pour positionner un nouveau mode, utilisez la commande **`setenforce`** :

Terminal

```
# setenforce permissive
```

Dans ce cas où le mode en cours est le mode permissif, mais au redémarrage de la machine, c'est le mode présent dans **`/etc/selinux/config`** qui sera pris en compte. Pour rendre ce mode permanent, il faut mettre à jour ce fichier en passant **`permissive`** à la directive SELINUX.

Il est également possible de voir le statut général de SELinux en appelant la commande **`sestatus`** :

Terminal

```
# sestatus
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             targeted
Current mode:                   permissive
Mode from config file:          enforcing
Policy MLS status:              enabled
Policy deny_unknown status:     allowed
Max kernel policy version:      28
```

La deuxième ligne **`SELinuxfs mount : /sys/fs/selinux`** montre que SELinux, lorsqu'il est activé, est chargé en mémoire en tant que système de fichiers de type **SELinuxfs**. Le fichier **`/etc/fstab`** contient une entrée pour le montage de SELinux :

Terminal
```
# cat mount| grep selinux
selinuxfs on /sys/fs/selinux type selinuxfs (rw,relatime)
```

On peut alors modifier à chaud l'état de SELinux par :

Terminal
```
# echo 0 > /sys/fs/selinux/enforce
```

NOTE

Il est préférable d'adopter le mode permissif au lieu de la désactivation complète de SELinux. En effet, en mode permissif, le noyau assure automatiquement la maintenance des étiquettes du système de fichiers SELinux appropriées, évitant ainsi un coûteux réétiquetage du système de fichiers lors du démarrage du système et lorsque SELinux est activé.

4.3 Activation/désactivation de SELinux depuis GRUB

Il est possible d'agir sur l'état de SELinux lors du démarrage du système en passant des paramètres au noyau Linux depuis la configuration du GRUB (Legacy ou GRUB 2). Dans ce cas, trois paramètres peuvent être affectés :

1. `selinux=0` : pour informer le système de la désactivation complète de SELinux. Cette action est similaire au positionnement de la valeur `SELINUX=disabled` dans le fichier de configuration de SELinux ;
2. `enforcing=0` : pour informer le système d'exécuter SELinux en mode permissif. Cette action est similaire au positionnement de la valeur `SELINUX=permissive` dans le fichier de configuration de SELinux ;
3. `enforcing=1` : pour informer le système d'exécuter SELinux en mode strict. Cette action est similaire au positionnement de la valeur `SELINUX=enforcing` dans le fichier de configuration de SELinux.

Dans GRUB 2, il faut passer l'un de ces paramètres dans le fichier `/etc/default/grub` :

Fichier
```
GRUB_TIMEOUT=10
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="crashkernel=auto rhgb quiet enforcing=0" //Adopter le
mode permissif
GRUB_DISABLE_RECOVERY="true"
```

Pour activer ce paramètre, on lance la commande suivante :

Terminal
```
# grub2-mkconfig -o /boot/grub2/grub.cfg
```

5. EXEMPLE AVEC LE SERVEUR WEB APACHE

Dans ce paragraphe, nous allons découvrir le comportement de SELinux lors d'un accès non autorisé par la politique de sécurité. Comme exemple, nous allons changer le contexte SELinux du répertoire `/var/www/html` (voir section 4.1).
La commande `chcon` permet de réaliser cette opération :

Terminal

```
# chcon -t user_home_t -R /var/www/html
# ls -lZ /var/www/html
drwxr-xr-x. root root unconfined_u:object_r:user_home_t:s0 blog
# setenforce 1
```

Une fois que le contexte de sécurité a été changé, j'ai lancé **Firefox** pour accéder à mon hôte virtuel déjà créé. Le navigateur affiche un message d'erreur indiquant qu'il est impossible d'accéder à la racine de notre hôte virtuel (voir figure 5). Pourquoi ? C'est le fichier journal de SELinux qui répondra à cette question.

Figure 4

Terminal

```
# tail /var/log/audit/audit.log|grep httpd
type=AVC msg=audit(1467815427.735:2193): avc:  denied  { read } for
pid=34909 comm="httpd" name="index.html" dev="sda3" ino=1780501 scontext=s
ystem_u:system_r:httpd_t:s0 tcontext=unconfined_u:object_r:user_home_t:s0
tclass=file
type=SYSCALL msg=audit(1467815427.735:2193): arch=c000003e syscall=2
success=no exit=-13 a0=7f884d5162b0 a1=80000 a2=0 a3=4 items=0 ppid=34886
pid=34909 auid=4294967295 uid=48 gid=48 euid=48 suid=48 fsuid=48 egid=48
sgid=48 fsgid=48 tty=(none) ses=4294967295 comm="httpd" exe="/usr/sbin/
httpd" subj=system_u:system_r:httpd_t:s0 key=(null)
```

Cette sortie nécessite une autopsie :

⇨ `type=AVC` : il s'agit d'une violation de la stratégie de sécurité SELinux. Le message est de type **AVC** (*Access Vector Cache*). L'AVC est un sous-système de SELinux dans le noyau Linux. C'est ce dernier qui vérifie et charge les règles de la stratégie de sécurité et prend les décisions adéquates (permissions/refus). Tous les accès sont mis en cache dans l'AVC, d'où le nom de « Cache », pour une consultation ultérieure. Tout accès futur est vérifié depuis l'AVC : si une permission particulière ne se trouve pas dans le cache, alors c'est la politique de sécurité SELinux qui sera consultée de nouveau ;

- **avc: denied. {read}** : c'est l'action SELinux. Dans notre situation, SELinux a refusé l'accès en lecture ;
- **pid=34909** : l'identifiant du processus qui a généré l'action est **34909** ;
- **comm="httpd"** : la commande sans argument qui est à l'origine de l'action ;
- **name="index.html"** : l'objet cible sollicité en lecture par le processus **httpd** ;
- **dev="sda3"** : l'objet cible (**index.html**) réside dans la partition **/dev/sda3** ;
- **ino=1780501** : le numéro d'*inode* de l'objet cible ;
- **scontext=system_u:system_r:httpd_t:s0** : c'est le contexte de sécurité du processus source qui est dans le cas présent **httpd** ;
- **tcontext=unconfined_u:object_r:user_home_t:s0** : le contexte de sécurité de l'objet cible (toujours **index.html**) ;
- **tclass=file** : c'est la classe de l'objet cible qui peut être un répertoire, un fichier, une *socket* tcp/udp, un node, un descripteur de fichier, un *pipe*, un lien symbolique, etc. Ici la classe est de type fichier (**file :index.html**).

La suite du message est de **type=SYSCALL** : il se produit en même temps que celui du type AVC et il s'agit de l'appel système qui a déclenché le message d'audit AVC.

Finalement, comment peut-on traduire le message d'audit précédent du langage SELinux vers le langage humain ? SELinux a bloqué (*denied*) l'opération de lecture (*read*) du fichier **index.html**, ayant un numéro d'*inode* **178051** et qui réside sur **/dev/sda3**, opération demandée par le processus **httpd** (PID=**34909**). Le processus **httpd** s 'exécute avec un contexte de sécurité : **system_u:system_r:httpd_t** alors que le fichier **index.html** possède le contexte **SELinux unconfined_u:object_r:user_home_t**. Cette cible (**index.html**) est classée par SELinux en tant que **file**.

6. CONTRÔLE DES PORTS RÉSEAUX AVEC SELINUX

Le contrôle d'accès de type MAC ne se limite pas uniquement aux contrôles des fichiers et aux processus, mais il est également capable de contrôler les communications réseaux. En effet, ce mécanisme de contrôle d'accès est basé principalement sur les accès aux *sockets* ; les permissions SELinux de type **name_bind** et **name_connect** seront mises en jeu.

Dans la pratique, il s'agit de déterminer sur quel *socket* un processus est capable d'accepter les connexions et sur quel *socket* un client peut éventuellement se connecter. Les *sockets* sont soit de type TCP, soit de type UDP. Ces *sockets* sont classifiés par SELinux respectivement par **tcp_socket** et **udp_socket**.

Toujours avec le serveur web Apache, qui est normalement en écoute sur les ports **80** et **443** pour le https, nous allons changer le port **80** par un port non régulier choisi par hasard, par exemple **82**. Ceci se fait dans le fichier de configuration **httpd.conf** via la directive **LISTEN**. Redémarrons le serveur **httpd** :

Terminal
```
# systemctl restart httpd.service
Job for httpd.service failed because the control process exited with
error code. See "systemctl status httpd.service" and "journalctl -xe" for
details.
```

La première constatation est que le démon **httpd** n'a pas pu démarrer :

Terminal
```
# systemctl status httpd.service
httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; disabled; vendor
preset: disabled)
   Active: failed (Result: exit-code) since Sat 2016-07-09 18:36:01 GMT;
4min 38s ago
```

Deuxièmement, une erreur est reportée dans le journal d'audit de SELinux :

Terminal
```
# tail /var/log/audit/audit.log
type=AVC msg=audit(1468089361.821:2316): avc:  denied  { name_bind } for
pid=37700 comm="httpd" src=82 scontext=system_u:system_r:httpd_t:s0 tconte
xt=system_u:object_r:reserved_port_t:s0 tclass=tcp_socket.
```

On interprète le message d'erreur de la façon suivante : le domaine (processus) **httpd_t** a essayé de créer une connexion (**name_bind**) via une *socket* TCP (**tclass=tcp_socket**) ayant le contexte **reserved_port**. Cette tentative de connexion est bloquée par SELinux (**avc:denied**). Il est indiqué dans le message d'audit un numéro de port **82** (**src=82**).

En effet, pour SELinux, initialiser une connexion TCP sur le port **82** par le serveur web Apache sort de l'ordinaire, puisque les ports dédiés au protocole HTTP en général sont connus par SELinux, sauf exception passée par une règle modifiant la stratégie par défaut.

Pour afficher les ports dédiés à tout processus ayant un contexte SELinux **httpd_t** :

Terminal
```
# semanage port -l | grep httpd_port
http_port_t           tcp      80, 81, 443, 488, 8008, 8009, 8443, 9000
pegasus_http_port_t   tcp      5988
```

Donc notre serveur web est autorisé à ouvrir une connexion TCP uniquement via les ports indiqués ci-dessus. Le port **82** ne fait pas partie de cette liste, SELinux avait raison.

La commande suivante affiche la règle définie dans SELinux pour gérer ce cas de figure :

Terminal
```
# sesearch -s httpd_t -t http_port_t -A
allow httpd_t http_port_t : tcp_socket name_bind ;
allow httpd_t http_port_t : udp_socket name_bind ;
allow httpd_t http_port_t : tcp_socket name_connect ;
allow httpd_t http_port_t : tcp_socket name_connect ;
```

En cas de besoin, il est possible de changer le contexte d'un port particulier. Si on a besoin de prendre en charge le port TCP **82** comme étant un port dédié au http, une simple ligne de commandes suffit, encore avec **semanage** :

Terminal
```
# semanage port -a -t http_port_t -p tcp 82
```

Vérifions l'ajout de ce port :

Terminal
```
# semanage port -l |grep http_port
http_port_t  tcp      82, 80, 81, 443, 488, 8008, 8009, 8443, 9000
```

Le port **82** est maintenant premier sur la liste.

NOTE

La commande `semanage` fait partie intégrante du paquetage `policycoreutil-python`.

Le serveur web démarre sans soucis après la mise à jour de la politique de sécurité SELinux :

Terminal
```
# systemctl status httpd.service
● httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; disabled; vendor
preset: disabled)
   Active: active (running) since Wed 2016-07-13 03:33:43 GMT; 5s ago
```

Vérifions le port d'écoute :

Terminal
```
# netstat -patune | grep httpd
tcp6    0    0 :::82      :::*      LISTEN      0      296224    39603/httpd
tcp6    0    0 :::443     :::*      LISTEN      0      296228    39603/httpd
```

Tout est bon ! Le serveur web écoute désormais sur le port **82** et sur le port **443**.

7. LES BOOLÉENS SELINUX

La politique de sécurité SELinux comporte un ensemble de règles (environ 304 règles) permettant de modifier la stratégie par défaut sans toucher à la politique originale. Cette méthode permettant d'agir sur la politique facilite énormément l'administration de SELinux et les administrateurs de la sécurité, en utilisant ces règles, peuvent affiner la politique afin de procéder à des ajustements sélectifs. Ces règles sont appelées les booléens. Les booléens SELinux sont des commutateurs qui modifient le comportement de la stratégie SELinux. Ce sont des règles que l'on peut activer ou désactiver à tout moment.

Pour afficher les booléens SELinux et leur valeur actuelle, on utilise cette commande :

Terminal
```
# getsebool -a
ftp_home_dir --> on
ftpd_anon_write --> off
ftpd_connect_all_unreserved --> off
ftpd_connect_db --> off
ftpd_full_access --> off
ftpd_use_cifs --> off
```

```
ftpd_use_fusefs --> off
ftpd_use_nfs --> off
ftpd_use_passive_mode --> off
httpd_enable_cgi --> on
httpd_enable_ftp_server --> off
httpd_enable_homedirs --> off
httpd_execmem --> off
httpd_graceful_shutdown --> on
httpd_manage_ipa --> off
httpd_mod_auth_ntlm_winbind --> off
httpd_mod_auth_pam --> off
httpd_read_user_content --> off
```

La sortie a été tronquée, mais comme indiqué plus haut il existe 304 booléens dans la version actuelle de la politique SELinux.

Revenons à l'exemple du serveur web de la section 5 : lorsque le contexte de sécurité du répertoire **/var/www/html** est changé en **user_home_t**, SELinux a bloqué l'accès aux fichiers et l'hôte virtuel n'est plus desservi. En effet, le serveur web ne pourra pas lire les fichiers se trouvant dans les répertoires personnels des utilisateurs, c'est le comportement par défaut de SELinux. Pour changer ce comportement, les booléens sont le plus simple recours. Il faut donc changer l'état d'un booléen particulier permettant de débloquer ce cas. Ce booléen est **httpd_read_user_content**, qui doit basculer de **off** à **on**.

La commande **setsebool** réalisera l'action :

Terminal

```
# setsebool  httpd_read_user_content on
```

C'est tout ! L'hôte virtuel est désormais accessible.

Cette modification n'est pas permanente : lors d'un redémarrage, on perd ce changement. Pour rendre le changement d'un booléen persistant, on ajoute **-P** à la commande **setsebool** :

Terminal

```
# setsebool -P  httpd_read_user_content on
```

La commande **semanage boolean -l** indique si un booléen est permanent ou pas, et permet d'afficher une brève description de celui-ci :

Terminal

```
# semanage boolean -l | grep httpd_read_user_content
httpd_read_user_content        (on   ,   on)  Allow httpd to read user content
```

8. DÉPANNAGE DE SELINUX

Que faire lorsque SELinux bloque l'accès aux fichiers ou bloque l'accomplissement d'une tâche par un processus ? Il n'y a pas une réponse absolue à ceci, mais tout dépend de la situation et chaque cas est un cas particulier. Cependant, avant de commencer à procéder à des ajustements quelconques, envisagez que SELinux fait peut-être son travail correctement en interdisant la tentative d'accès. Comme vu dans ce qui précède, si un serveur web tente d'accéder à un fichier ayant un contexte différent de ce qu'il prétend être, ce peut être un signe que le service a été compromis. Autrement, si vous envisagez que tout de même l'accès devait être accordé, il convient de prendre d'autres mesures pour résoudre le problème.

Disons que le problème SELinux le plus courant est un contexte de fichier incorrect. Ce cas peut survenir lorsqu'un fichier est créé à un emplacement avec un contexte de fichier

spécifique, puis déplacé dans un endroit ayant un contexte de sécurité différent. Dans ce cas, changer le contexte de sécurité d'un tel fichier pour résoudre le problème.

Heureusement SELinux fournit le paquetage **setroubleshoot-server** permettant d'analyser et de surveiller les infractions SELinux. En effet, ce paquet retransmet les messages d'audit de **/var/log/audit/audit.log** vers **/var/log/messages** en envoyant un bref résumé de l'infraction. Ce résumé comprend également les identificateurs uniques (UUID) des infractions SELinux. Ces identificateurs servent de base pour l'identification des éventuels problèmes et à la génération d'un rapport de résolution spécifique pour un incident particulier. C'est la commande **sealert** qui produit ce rapport d'incident.

Revenons à notre cas de test, le serveur web. Lorsqu'on a changé le contexte de sécurité de **/var/www/html**, voici le message d'audit envoyé par SELinux :

Fichier

```
type=AVC msg=audit(1467847794.417:2298): avc:  denied  { read } for
pid=34910 comm="httpd" name="index.html" dev="sda3" ino=1780501 scontext=s
ystem_u:system_r:httpd_t:s0 tcontext=unconfined_u:object_r:user_home_t:s0
tclass=file
```

Inspecter en même temps le fichier **/var/log/messages** peut révéler quelques informations complémentaires :

Fichier

```
SELinux is preventing /usr/sbin/httpd from read access on the file /var/
www/html/blog/index.html. For complete SELinux messages. run sealert -l
84ceeeb3-3d39-4ecb-8011-48a40c0ed7e8
```

Il est indiqué l'UUID de l'infraction : **84ceeeb3-3d39-4ecb-8011-48a40c0ed7e8**. Ainsi, avec la commande **sealert** on peut en déduire une résolution adéquate à notre problème :

Terminal

```
# sealert -l 84ceeeb3-3d39-4ecb-8011-48a40c0ed7e8 > incident
# cat incident
```

Fichier

```
SELinux is preventing /usr/sbin/httpd from read access on the file /var/
www/html/blog/index.html.

*****  Plugin restorecon (92.2 confidence) suggests   *****
If you want to fix the label.
/var/www/html/blog/index.html default label should be httpd_sys_content_t.
Then you can run restorecon.
Do
# /sbin/restorecon -v /var/www/html/blog/index.html

*****  Plugin catchall_boolean (7.83 confidence) suggests   *****
If you want to allow httpd to read user content
Then you must tell SELinux about this by enabling the 'httpd_read_user_
content' boolean.
You can read 'user_selinux' man page for more details.
Do
setsebool -P httpd_read_user_content 1

*****  Plugin catchall (1.41 confidence) suggests   *****
If you believe that httpd should be allowed read access on the index.html file
by default.
Then you should report this as a bug.
You can generate a local policy module to allow this access.
Do
```

```
allow this access for now by executing:
# grep httpd /var/log/audit/audit.log | audit2allow -M mypol
# semodule -i mypol.pp

Raw Audit Messages
type=AVC msg=audit(1467847952.761:2309): avc:  denied  { read } for  pid=34911
comm="httpd" name="index.html" dev="sda3" ino=1780501 scontext=system_u:system
_r:httpd_t:s0 tcontext=unconfined_u:object_r:user_home_t:s0 tclass=file
```

Ce rapport est exhaustif, il y est indiqué l'ensemble des correctifs éventuels, et suivant le besoin on choisira la manière adéquate de corriger le problème. Dans notre cas, le changement du contexte de sécurité suffit :

Terminal
```
# /sbin/restorecon -r -v /var/www/html/
```

9. ACTIVER SELINUX DANS DEBIAN

Et pour finir cet article, les utilisateurs de Debian pourront également bénéficier de SELinux dans leurs distributions.

Le noyau Linux sous Debian supporte SELinux, mais il est non activé par défaut. Pour la mise en place de SELinux sous Debian, voici les étapes à suivre :

1 Les paquetages à installer :

Terminal
```
# apt-get install selinux-basics selinux-policy-default auditd
```

2 Activation de SELinux :

Terminal
```
# selinux-activate
```

Cette commande effectue la mise à jour de grub et des modules PAM et permet de créer le fichier `./autorelabel` pour étiqueter le système de fichiers.

3 Redémarrage nécessaire :

Terminal
```
# reboot
```

4 Vérifier que tout est en place :

Terminal
```
# check-selinux-installation
```

CONCLUSION

L'implémentation de SELinux dans le noyau Linux ajoute un composant fondamental pour la sécurité d'un système exécutant GNU/Linux. Simple à administrer et à mettre en place, SELinux restera un produit essentiel pour les responsables de sécurité afin de prévenir des tentatives d'intrusion ou encore de compromissions d'un système, sachant qu'il leur est également possible de rédiger leurs propres règles et de les personnaliser selon leurs besoins stratégiques. ■

ANNEXE

ANNEXE

Retrouvez dans cette partie les définitions de certains termes et acronymes présents dans les articles.

ANNEXE

INDEX

Nous vous proposons un récapitulatif des différents termes techniques et surtout des nombreux acronymes que vous pouvez rencontrer dans les différents articles de ce hors-série afin de ne pas être gênés dans votre lecture.

ALU

Unité chargée d'effectuer les calculs au sein d'un microprocesseur. ALU signifie **A**rithmetic-**L**ogic **U**nit.

Voir « Cloisonnement des processus au sein du noyau Linux » en page 40.

APIC

L'**A**dvanced **P**rogrammable **I**nterrupt **C**ontroller est un contrôleur programmable d'interruptions.

Voir « Interactions entre espace utilisateur, noyau et matériel » en page 22.

BIG ENDIAN

Notation dans laquelle l'ordre de stockage en mémoire d'un mot (au sens d'ensemble d'octets) se fait de la gauche vers la droite (l'octet de poids le plus fort en premier). La notation big endian est l'inverse de la notation little endian.

Prenons par exemple la valeur hexadécimale suivante `AA 12 5F 8C`, sera stockée sous la forme `AA 12 5F 8C`.

La notation big endian est également appelée gros-boutiste.

Voir « Interactions entre espace utilisateur, noyau et matériel » en page 22.

BIOS

Le BIOS (**B**asic **I**nput **O**utput **S**ystem) est le programme contenu dans la ROM d'un ordinateur et s'exécutant au démarrage (avant le système d'exploitation donc dès la mise sous tension) et permettant d'effectuer toutes les opérations élémentaires.

Voir « Interactions entre espace utilisateur, noyau et matériel » en page 22.

CGROUPS

Les cgroups (pour **C**ontrol **Groups**) représentent une fonctionnalité intégrée au noyau permettant de contrôler l'utilisation des ressources d'un processus selon plusieurs critères.

Voir « Cloisonnement des processus au sein du noyau Linux » en page 40.

DEVICE TREE

Structure de données permettant de décrire le matériel.

Voir « Interactions entre espace utilisateur, noyau et matériel » en page 22.

FPU

Unité chargée d'effectuer les calculs en virgule flottante au sein d'un microprocesseur. FPU signifie **F**loating-**P**oint **U**nit.

HYPERVISEUR

Plateforme de virtualisation. Permet d'utiliser en même temps plusieurs systèmes d'exploitation au sein d'une même machine.

Les hyperviseurs sont classés en deux catégories :

- Type 1 : hyperviseur natif (ou *bare metal*) qui s'exécute directement sur le système. Par exemple, KVM est un hyperviseur de type 1.
- Type 2 : hyperviseur hébergé (ou *hosted*) qui s'exécute au sein d'un autre système d'exploitation. Par exemple, VirtualBox est un hyperviseur de type 2.

Voir « KVM : focus sur l'implémentation d'un hyperviseur dans Linux » en page 88.

IPC

La communication inter processus (**I**nter-**P**rocess **C**ommunication) regroupe les mécanismes de communication entre processus concurrents.

Voir « Cloisonnement des processus au sein du noyau Linux » en page 40.

IRQ

IRQ pour **I**nterrupt **Req**uest. Interruption matérielle déclenchée par un périphérique du système.

Voir « Interactions entre espace utilisateur, noyau et matériel » en page 22.

ISR

Une ISR (pour **I**nterrupt **S**ervice **R**outine) est un sous-programme appelé lors d'une interruption.

Voir « Interactions entre espace utilisateur, noyau et matériel » en page 22.

KVM

KVM pour **K**ernel-based **V**irtual **M**achine (ou machine virtuelle basée sur le noyau) est un hyperviseur (voir définition dans le présent glossaire) intégré dans le noyau GNU/Linux.

Voir « KVM : focus sur l'implémentation d'un hyperviseur dans Linux » en page 88.

ANNEXE

LITTLE ENDIAN

Notation dans laquelle l'ordre de stockage en mémoire d'un mot (au sens d'ensemble d'octets) se fait de la droite vers la gauche (l'octet de poids le plus faible en premier). La notation little endian est l'inverse de la notation big endian.

Prenons par exemple la valeur hexadécimale suivante `AA 12 5F 8C`, sera stockée sous la forme `8C 5F 12 AA`. La notation little endian est également appelée petit-boutiste.

Voir « Interactions entre espace utilisateur, noyau et matériel » en page 22.

LSM

Linux **S**ecurity **M**odule est une infrastructure intégrée au noyau GNU/Linux définissant des modèles formels de sécurité.

Voir « La sécurité du noyau avec SELinux » en page 104.

LXC

Système de virtualisation dont le nom provient de **LinuX C**ontainers. Permet d'isoler l'exécution des applications.

Voir « Cloisonnement des processus au sein du noyau Linux » en page 40.

MMU

MMU pour **M**emory **M**anagement **U**nit soit unité de gestion mémoire. Composant contrôlant les accès à la mémoire du processeur.

Voir « Cloisonnement des processus au sein du noyau Linux » en page 40.

MUTEX

Exclusion mutuelle (Mutex pour **Mut**ual **ex**clusion) : permet de synchroniser les ressources du système, d'éviter des accès en même temps à des ressources partagées.

Voir « Interactions entre espace utilisateur, noyau et matériel » en page 22.

NAMESPACE

Les namespaces ou espaces de noms permettent de limiter les interactions entre processus (regroupement d'un ensemble de processus selon un ou plusieurs critères).

Voir « Cloisonnement des processus au sein du noyau Linux » en page 40.

PID

Identifiant de processus (PID pour **P**rocess **ID**entifier). Le premier processus démarré sous GNU/Linux est `init` et il a logiquement comme PID `1`.

GNU

COMPRENDRE, UTILISER ET ADMINISTRER LINUX

LINUX PRATIQUE

ESSENTIEL

À partir du 23 décembre,

Linux Pratique change...

...découvrez sa

Nouvelle formule !

+ de Raspberry Pi

+ de tutoriels

+ d'initiation à la programmation

+ de logithèque

Les nouvelles rubriques de votre magazine :

- Cahier Raspberry Pi & débutant Linux
- Programmation & scripts
- Logithèque & applicatif
- Mobilité & objets connectés
- Système & personnalisation
- Web & réseau
- Terminal & ligne de commandes
- Entreprise & organisation
- Réflexion & société ...

COMPRENEZ, UTILISEZ & ADMINISTREZ LINUX SUR PC, MAC & RASPBERRY PI AVEC LINUX PRATIQUE !

ANNEXE

PROCESSUS

Un processus est une tâche/programme en cours d'exécution sur un système.

QEMU

Logiciel de virtualisation utilisant l'hyperviseur KVM.

ROOTKIT

Ensemble de logiciels malveillants (le « kit » de rootkit) permettant de mettre en place un accès non autorisé en général en tant que root (le « root » de rootkit) à une machine, le tout de manière furtive.

Voir « Modification des appels systèmes du noyau Linux : manipulation de la mémoire du noyau » en page 54.

SELINUX

Module de sécurité (SELinux vient de **SE**curity-enhanced **Linux** soit Sécurité renforcée pour Linux). Il s'agit d'un LSM (voir définition dans le présent glossaire).

Permet à l'administrateur de définir une politique de sécurité (renforce la sécurité du système en se basant sur des règles de sécurité).

Voir « La sécurité du noyau avec SELinux » en page 104.

SPINLOCK

Le spinlock (verrou tournant) est un mécanisme permettant d'empêcher les accès concurrents à un buffer contenant des données stockées par une routine d'interruption.

Voir « Interactions entre espace utilisateur, noyau et matériel » en page 22.

UTS HOSTNAME

UNIX **T**imesharing **S**ystem **Hostname** : il s'agit de l'une des six *namespaces*.

Voir « Cloisonnement des processus au sein du noyau Linux » en page 40.

VFS

Le **V**irtual **F**ile **S**ystem est la dénomination qui regroupe les différents systèmes de fichiers virtuels sous Linux (et UNIX dans un sens plus large). Un VFS permet d'utiliser de manière transparente plusieurs systèmes de fichiers.

Voir « Généralités sur le noyau Linux » en page 08. ▮

VISITEZ NOTRE NOUVELLE BOUTIQUE ET DÉCOUVREZ NOS GUIDES !

LINUX MAGAZINE / FRANCE | GNU LINUX PRATIQUE ESSENTIEL | MISC Multi-System & Internet Security Cookbook | HACKABLE MAGAZINE | Open Silicium

ET VOUS ?
COMMENT LISEZ-VOUS VOS MAGAZINES PRÉFÉRÉS ?

« Moi, je les lis en version PAPIER ! »

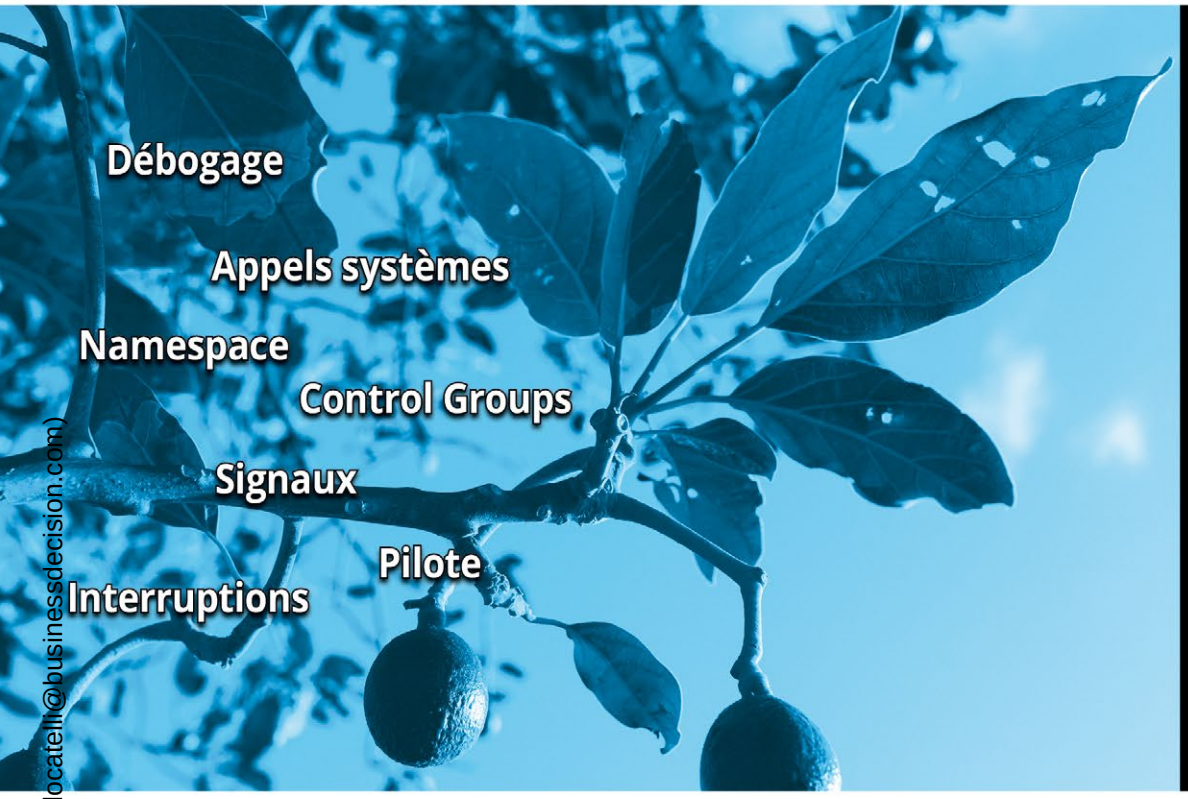
« Moi, je les lis en version PDF ! »

« Moi, je consulte la BASE DOCUMENTAIRE ! »

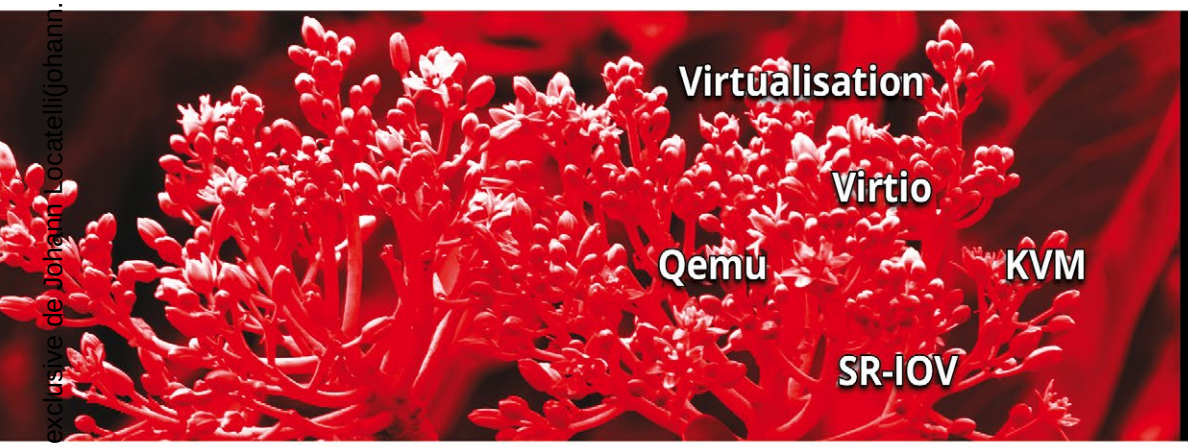
RENDEZ-VOUS SUR **www.ed-diamond.com**

POUR DÉCOUVRIR TOUTES LES MANIÈRES DE LIRE VOS MAGAZINES PRÉFÉRÉS !

DÉBUTEZ
Recompilez et adaptez le noyau à votre PC



EXPÉRIMENTEZ
5 cas concrets de développement noyau



VIRTUALISEZ
Étudiez le fonctionnement de l'hyperviseur KVM

SÉCURISEZ
Utilisez SELinux pour sécuriser l'accès à vos services